Locality Optimizations for Regular and Irregular Applications

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Samyam Rajbhandari, B.A., M.Sc.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2016

Dissertation Committee:

P. Sadayappan, Advisor
Louis-Noel Pouchet
Srinivasan Parthasarathy
Abstract

The fastest supercomputer in the world as of July 2016 is the Sunway TaihuLight. It can achieve a staggering performance of 93 PetaFlops. This incredible performance is achieved via massive parallelism. Today’s supercomputers and compute clusters have tens of thousands of distributed memory nodes with each node comprised of several shared memory multi/many core processors.

Scaling on these massively parallel systems is not an easy task. A major performance and scalability bottleneck is the limited data movement bandwidth, which can be orders of magnitude smaller than the computation bandwidth. Developing applications to scale on these massively parallel systems requires minimizing data movement volume at different levels of memory hierarchy using locality optimization techniques. Locality optimization aims to reduce the data movement between slow and fast memory by rescheduling/remapping the original computation to reuse the data once it is in fast memory, thereby avoiding subsequent movement of the same data from slow memory.

This dissertation explores multiple aspects of locality optimizations for enhancing scalability and performance of various regular and irregular applications on massively parallel computing environment. It develops distributed algorithms, lower bound techniques, and compiler and runtime frameworks for optimizing Tensor Contractions, Four-Index Transform, Convolutional Neural Networks (CNNs), and Recursive Tree
Traversal on k-d trees. Each of these application domains is limited in performance and scalability primarily by data movement costs at a particular level of memory hierarchy.

To be specific, on a massively parallel system, distributed Tensor Contractions can have limited scalability due to the cost of communication between distributed memory nodes. The Four-Index Transform, on the other hand, can be limited in the size of largest problem that can be completed in a reasonable amount of time due to data transfer cost from disk to memory. On a multi-core CPU, the state-of-art approach for training CNNs can have limited scalability and performance due to relatively large data movement between L3/L2 cache. Similarly, Recursive Tree Traversal programs on k-d trees are limited in performance and scalability by memory/cache bandwidth.

This thesis develops solutions primarily for reducing the aforementioned data movement costs. The solutions developed in this thesis improves performance and scalability of the above mentioned computations resulting in overall speedups ranging from 4x to more than 10x over the state-of-art on target systems.
Acknowledgments

Pursuing the doctoral degree has been an intellectual and a spiritual journey for me. I am grateful to many people for their love, support, and guidance, without which I would be lost.

First and foremost, I would like to express my earnest gratitude towards my advisor Dr. P. Sadayappan. His compassion knows no bounds. In the early years of my Ph.D. I was struggling to recover from tendonitis on my wrists which made it difficult for me to use my hands. It was his patience and encouragement at this time that allowed me to continue. Since then, he has been a constant source of guidance and inspiration. I will forever be indebted to him for all he has taught me.

In addition to my advisor, I have had the privilege to work with many great minds in the last five years. This dissertation would be nowhere near its quality if it weren’t for the guidance from Sriram Krishnamoorthy, Fabrice Rastello, Yuxiong He, Robert J. Harrison, Louis-Noel Pouchet, and Karol Kowalski. They have mentored me at various stages of my Ph.D., helped me refine my ideas, and pushed me to discover answers when I thought there were none.

I am grateful to Srinivasan Parthasarathy for agreeing to be part of my dissertation committee. I am thankful to my colleagues Kevin Stock, Pai-Wei Lai, Abhijit Vageeswara, Jinsung Kim and Martin Kong for their help with various projects. I am especially thankful to Akshay Nikam for his friendship, and for the countless late
night hours we spent working together. I am also grateful to my lab-mates Venmugil Elango, Sanket Traverigi, Aarash Ashari, Prashant Rawat, and Tom Henretti for their pleasant company in the lab.

The Ohio State University has truly been a home away from home for me. For this, I thank my dear friends Sidharth Mohan, Bishal Karna, Pranietha Mudliar, Nikhil Nair, Navni Verma, Ania Stakun, Sagar Sharma, Dima Kondratjev, Bettina Barallis, Sanjay Ramdon, Priyanka Shetty and Travis Jones. I am grateful to them for their love and support, and for the countless days and nights filled with laughter and joy.

Last but not least, I would like to thank my family. Thank you, 'Di’ for instilling infinite curiosity in me, and for showing me how through logic and reason, even the deepest of mysteries could be unraveled. Thank you, ‘Nanan’ for showering me with unbounded love, and for teaching me how to be caring, kind, and compassionate even in the most difficult of times. Thank you, Shreya. You are the best brother I could ask for. Thank you for sharing your passion for life with me, for partaking in my adventures, and for supporting and inspiring me from time to time with your own actions and dreams. And finally thank you, Olia. You have been my knight in shining armor. It is knowing that you are by my side that has made this journey a joyful one, even at the most difficult of times.

Samyam Rajbhandari

June 22, 2016
Vita

June, 2011 .................................. B.A. with Honors in Physics, Williams College, Williamstown, MA 01267, USA
June-August 2014 ......................... Visiting Scholar, Computational Science Center, Brookhaven National Labs, Upton, NY, USA
August 2014 ............................... M.Sc. in Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
Feb-August 2016 ........................... Research Intern, Microsoft Research, Redmond, WA, USA
August 2012 - Present ..................... Graduate Research Associate, The Ohio State University,

Publications

Research Publications

Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, Robert J. Harrison, P. Sadayappan “A Domain-Specific Compiler for a Parallel Multiresolution Adaptive Numerical Simulation Environment”. International Conference for High Performance Computing, Networking, Storage and Analysis, Nov 2016 (accepted with Shepherding)


vi


**Fields of Study**

Major Field: Department of Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xv</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. A Communication-Optimal Framework for Contracting Distributed Tensors</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Preliminaries</td>
<td>13</td>
</tr>
<tr>
<td>2.2.1 Definitions and Notations</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Iteration Space and Data Space Mapping</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1 Categorization of input tensor distribution</td>
<td>19</td>
</tr>
<tr>
<td>2.3.2 Problem Statement</td>
<td>20</td>
</tr>
<tr>
<td>2.4 RRR framework</td>
<td>20</td>
</tr>
<tr>
<td>2.4.1 Reuse Directions</td>
<td>21</td>
</tr>
<tr>
<td>2.4.2 Compatible Input Distribution</td>
<td>22</td>
</tr>
<tr>
<td>2.4.3 Compatible Output Distribution</td>
<td>25</td>
</tr>
<tr>
<td>2.4.4 Fundamental Communication Operators:</td>
<td>26</td>
</tr>
<tr>
<td>Reduction, Recursive Broadcast and Rotation</td>
<td>26</td>
</tr>
<tr>
<td>2.4.5 Generating Contraction Algorithms</td>
<td>31</td>
</tr>
<tr>
<td>2.4.6 Dimension Scaling</td>
<td>32</td>
</tr>
<tr>
<td>2.4.7 Finding optimal mappings</td>
<td>34</td>
</tr>
</tbody>
</table>
## 4. Optimizing 4-index Transform

4.1 Introduction .................................................. 82
4.2 Background .................................................. 86
  4.2.1 NWChem and four-index transform ..................... 86
  4.2.2 Data movement lower bounds .......................... 91
4.3 Utility of Fusion ............................................. 94
  4.3.1 From tight lower bound to futility of fusion ........ 94
4.4 Analysis of fusion: four-index transform ................. 96
  4.4.1 Necessary condition: utility of fusion ............... 97
  4.4.2 Tight I/O bound for useful fusion ................... 98
  4.4.3 Fusion Choices: four-index transform ............... 100
4.5 Necessary and sufficient conditions for full reuse ...... 102
  4.5.1 Necessary conditions ................................. 102
  4.5.2 Sufficient condition ................................ 106
4.6 Improving the Four-index transform in NWChem .......... 108
  4.6.1 Largest zero-spill four-index transform ............ 108
  4.6.2 Minimizing communication volume .................... 109
  4.6.3 Mapping to processors ................................ 111
  4.6.4 Space-time trade-off due to symmetry breaking ...... 113
4.7 Experimental Results ........................................ 116
4.8 Conclusions .................................................. 118

## 5. Framework for Optimizing CNNs on CPUs

5.1 Introduction .................................................. 120
  5.1.1 Convolutional Neural Networks ....................... 121
  5.1.2 Performance Characterization ......................... 123
  5.1.3 Optimization Framework for CNNs .................... 124
  5.1.4 Contributions .......................................... 125
5.2 Background .................................................. 126
  5.2.1 Artificial Neural Networks (ANN) ..................... 126
  5.2.2 Convolutional Neural Network (CNN) .................. 128
  5.2.3 Executing CNNs ........................................ 130
5.3 Performance Characterization ............................... 131
  5.3.1 Single-Core Performance ............................. 131
  5.3.2 Multicore Scalability .................................. 133
  5.3.3 Goodput ................................................. 135
5.4 Optimization Framework ..................................... 137
  5.4.1 GEMM-in-Parallel ...................................... 137
5.4.2 Sparse-Kernel ........................................ 139
5.4.3 Stencil-Kernel ....................................... 143
5.4.4 Putting it all together ................................ 147
5.5 Experimental Evaluation ............................... 148
  5.5.1 Methodology ........................................ 148
  5.5.2 Improvements of Real World Convolutions ........... 149
  5.5.3 End-to-End evaluation on CIFAR-10 training ........ 150
5.6 Related Work .......................................... 152
5.7 Conclusion ............................................. 154

6. On Fusing Recursive Traversals of K-d Trees ......... 155
  6.1 Introduction .......................................... 155
  6.2 Background ........................................... 158
    6.2.1 k-d Trees ........................................ 158
    6.2.2 Operations on k-ary Trees ......................... 159
  6.3 FuseT Compiler Framework ............................ 160
    6.3.1 Primitive Operators ................................ 161
    6.3.2 Data Access Types ................................. 162
    6.3.3 Examples ........................................... 163
  6.4 Fusion in FuseT ...................................... 165
    6.4.1 The Computation Space of a FuseT Program ........ 166
    6.4.2 Recursive Specification of a Fused Schedule .... 167
    6.4.3 Ordering between Node Operations ................ 168
  6.5 When Does a Valid Fused Schedule Exist? ............ 169
    6.5.1 Dependences in Operator DAG ....................... 170
    6.5.2 Necessary and Sufficient Conditions ............... 171
    6.5.3 Necessary Condition ................................ 172
    6.5.4 Sufficient Condition ................................ 175
  6.6 Traversal Fusion ..................................... 181
    6.6.1 Generating a Fused Schedule for a Valid OpDAG .... 181
    6.6.2 Generating a Sequence of Valid OpDAGs ............ 182
  6.7 Evaluation ........................................... 187
    6.7.1 MADNESS .......................................... 187
    6.7.2 Experiments ......................................... 189
    6.7.3 Performance Comparison ............................ 190
  6.8 Related Work ......................................... 192
  6.9 Conclusion ............................................ 194
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Exhaustive list of data space mapping of input tensors</td>
</tr>
<tr>
<td>2.2</td>
<td>Requirements for Compatible Distribution of Input Tensors</td>
</tr>
<tr>
<td>2.3</td>
<td>Scalability of Contraction 1 for N=256 on BG/Q in Seconds</td>
</tr>
<tr>
<td>2.4</td>
<td>Iteration Space Map and Data Space Maps</td>
</tr>
<tr>
<td>2.5</td>
<td>Predicted time complexities for different mappings</td>
</tr>
<tr>
<td>3.1</td>
<td>List of Contractions</td>
</tr>
<tr>
<td>4.1</td>
<td>Sizes of intermediate tensors. $s$ is the reduction due to spatial symmetry in tensor $C$.</td>
</tr>
<tr>
<td>4.2</td>
<td>Execution time (seconds) of four-index transform for tce_hyperpolar_ccsd_big on RI</td>
</tr>
<tr>
<td>4.3</td>
<td>Execution time (seconds) of four-index transform for Uracil using cc-pVQZ basis-set on RI</td>
</tr>
<tr>
<td>4.4</td>
<td>Execution time (seconds) of four-index transform for Uracil using cc-pVQZ basis-set on Cascade</td>
</tr>
<tr>
<td>5.1</td>
<td>Different Convolutions, their intrinsic AIT, the AIT corresponding to Unfold+GEMM, and the region in Fig. 5.1 that they belong to. These benchmarks were chosen to represent convolutions with high, moderate and low AIT. For varying levels of sparsity, these six benchmarks cover the entire performance characterization space shown in Fig. 5.1.</td>
</tr>
</tbody>
</table>
6.1 This table shows sets representing sequence of dependences (DS), and regular expressions for recognizing them. .......... 173

7.1 Performance Characteristics of m2 and M1 .......... 231
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Data space and iteration space of matrix multiplication as shown in Eq. 3.2 and Lst. 3.1.</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Iteration space mappings for 3-dimensional iteration space of matrix multiplication on to a 3-dimensional torus. In Map1 loop iterators I, J and K are distributed along torus dimensions ( p_0 ), ( p_1 ) and ( p_2 ) respectively. In Map2, loop iterators I and J are distributed along torus dimensions ( p_0 ) and ( p_2 ) respectively. Loop iterator K is serialized and the iteration space is replicated along torus dimension ( p_1 ), which means that each processor along ( p_1 ) computes the same set of points in the iteration space.</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Iteration space and data space mapping for matrix multiplication on a 2D torus, where ( p_I ) is reuse dimension for ( B ) and ( p_J ) is reuse dimension for ( A ).</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>Scheme for deriving contraction algorithm and input distribution from iteration space mapping.</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>a) DDO matrix multiplication. The double arrows along ( p_x ) and ( p_y ) show the dimension of the broadcast for ( k_A ) and ( k_B ) respectively. b) DDA vector product. Vector ( B[k] ) and ( C[i] ) are replicated along ( p_y ) and ( p_x ) respectively. The single arrow along ( p_x ) shows the dimension of reduction.</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>a) DC+DDA matrix multiplication. Both ( i ) and ( j ) are distributed along ( p_y ) and ( k_A ), and ( k_B ) is distributed along ( p_x ). Hence, ( B ) is rotated along ( p_y ) and partial results are reduced along ( p_x ). b) 2.5D SUMMA DDA+DDO matrix multiplication. ( k_2 ) is DDO along ( p_x ) and ( p_y ) hence broadcast is done along these dimensions. ( k_1 ) is DDA along ( p_z ) and hence reduction is done along ( p_z ).</td>
<td>30</td>
</tr>
</tbody>
</table>
2.7 Scalability of contractions 1, 3, 4 and 5 with \( N = 384 \) and contraction 2 with \( N = 64 \) on up to 16,384 nodes (262,144 cores) on BG/Q.

2.8 a) Actual vs Predicted on 256 BG/Q nodes. b) Actual vs Predicted on 256 Cray XE6 nodes.

3.1 Data space and iteration space of matrix multiplication as shown in Eq. (3.2) and Lst. 3.1.

3.2 Iteration space mappings for 3D iteration space of matrix multiplication on a 3D torus. In Map1, loop iterators I, J, and K are distributed along torus dimensions \( p_0 \), \( p_1 \), and \( p_2 \) respectively. In Map2, loop iterator I and J are distributed along torus dimensions \( p_0 \) and \( p_2 \), respectively. Loop iterator K is serialized, and the iteration space is replicated along torus dimension \( p_1 \), which means that each processor along \( p_1 \) computes the same set of points in the iteration space.

3.3 (a) \( C[i,j] = A[i,k]B[k,j] \) DDO Matrix Multiplication. The double arrows along \( p_x \) and \( p_y \) show the dimension of the broadcast for \( k_A \) and \( k_B \), respectively.

3.4 Block-Cyclic Distribution

3.5 Communication Pattern

3.6 Virtual splitting of a grid into 2D planes along the replicated dimension.

3.7 Broadcast groups of the form \( P[I,*,K] \).

3.8 Redistribution of \( A(i') \) from \( i \to x \) to \( i \to y \): the left figure shows the redistribution scheme described in this section while the right figure shows the naïve redistribution scheme.

3.9 The left figure shows the communication and computation time for seven contractions on 16,384 BG/Q nodes (262,144 cores), while the right figure shows times on 1,024 Cray XE6 nodes (24,576 cores).

3.10 Scalability of Contractions 2, 3, 6, and 7 up to 16,384 BG/Q nodes for \( N=256, 384, \) and 32,768.
3.11 Scalability of Contractions 2, 3, 6, and 7 up to 1024 Cray XE6 nodes for N=192, 256, and 32768. .......................................................... 77

4.1 Matrix-matrix multiplication kernel: untiled (top) and tiled (bottom) 92

5.1 CNN performance characteristics of using Unfold+Parallel-GEMM as a function of AIT (≈ 2× Number of features) and sparsity with respect to scalability, goodput and single core performance. The design space is divided into six regions. The figure shows their different performance characteristics and the optimization techniques of our work (depicted in bold) for improving them. In addition, it illustrates regions that real world image classification benchmarks (ImageNet-22k, ImageNet-1K, CIFAR-10, MNIST) occupy in the convolution space. ......................... 122

5.2 (a) An example of a 2D convolution using a 2 × 2 kernel with 2 input features of size 3 × 3 and 2 output features. (b) Unfolding a 2D image with two channels for convolving with a 2 × 2 convolution kernel. (c) MM $O = W \cdot U^T$ representing a 2D convolution using a 2 × 2 kernel with 2 output features. .................................................. 127

5.3 (a) Scalability of Parallel-GEMM on up to 16 cores for convolutions shown in Table. 5.1. (b) Sparsity across multiple epochs for three CNN benchmarks. ......................... 134

5.4 The set of figures shows absolute performance for dense convolutions, goodput for sparse convolutions, and relative speedups over baselines using each of the three techniques of $spg$-CNN. We present the results of 6 different convolutions in Table. 5.1. The results indicate that our techniques achieve significant performance improvement in Regions 2 (Fig. a, b) and 4 (Fig. c, d), and significant goodput improvement in Regions 1, 3, and 5 (Fig. e, f). ......................... 136

5.5 (a) Sparse matrix is first tiled along the columns, then each tile is stored in CT-CSR format. (b) A sparse column matrix is multiplied with a dense matrix to produce a dense column matrix as showing in Eq. 5.13. ......................... 140

5.6 Sparse Kernel for backward gradient calculations. Each arrow on the left represents a sparse-dense MM. The arrows on the right show the result’s storage location. ......................... 141
5.7 Performance improvement using our framework over Parallel-GEMM for convolution layers in real world benchmarks. For FP, the speedup of GEMM-in-Parallel over Parallel-GEMM is shown in blue, and if there is any, the additional speedup from Stencil-Kernel is shown in green. For BP, the speedup of Sparse-Kernel is shown in orange.

5.8 End-to-end performance comparison in CIFAR.

6.1 The figure shows a binary partition tree representing a 1D function. The figure also shows the region of 1D space that each leaf node of the tree represents.

6.2 This figure shows an OpDAG representing the sequence of operations given by Lst. 6.5. The blue arrows (labeled AD, SND and DD) shows dependences between OpCP and OpCC, and the green arrows (labeled f1, i1 and Output) shows the input and output fields to each of the operators.

6.3 a) The figure on the left shows the perf. improvement of fused Sum benchmark over the baseline for a varying length of the input vector. b) The figure on the right compares scalability between fused and baseline versions of Sum and OuterSum on up to 16 cores.

7.1 Binary spatial tree representing a 1D spatial function

7.2 X-MADNESS DSL Infrastructure

7.3 The figure shows the ordering between pre- and post- traversal operators enforced by fused traversal and sibling dependences. It also shows when a sequence of operators with sibling dependences can be fused, and when they cannot.

7.4 Strong Scalability for MatrixInner (m1)

7.5 Strong Scalability for Kinetic Energy Matrix Calculations (M1)

7.6 Strong and Weak Scalability for m2 Derivative Sequence (m2)
Listings

2.1 Matrix Multiplication ........................................ 15
2.2 Recursive Broadcast .......................................... 29
2.3 Rotate A .......................................................... 31
2.4 General tensor contraction algorithm .................... 32
3.1 Matrix Multiplication .......................................... 51
3.2 Recursive SUMMA ............................................... 55
3.3 General CAST ..................................................... 62
3.4 Redistribution Scheme ......................................... 71
4.1 Unfused 4-index transform ................................... 89
4.2 First two and last two fused ................................. 89
4.3 With re-computation ............................................ 89
4.4 Implementation of \( C[\alpha, j, k, l] \leftarrow A[i, j, k, l] \cdot B[\alpha, i] \) in NWChem 91
4.5 Tensor contraction schedule that achieves I/O equal to sum of its input and outputs .............................. 98
4.6 Fused pair of tensor contractions that achieves I/O equal to sum of size of its input and outputs .................. 100
4.7 Fused four-index transform that achieves I/O equal to the size of input and output ................................. 106
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.7</td>
<td>Inner Matrix operator</td>
<td>221</td>
</tr>
<tr>
<td>7.8</td>
<td>Inner Matrix operator after permutation</td>
<td>222</td>
</tr>
<tr>
<td>7.9</td>
<td>Final Inner Matrix operator</td>
<td>222</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Today, a small portable smartphone offers the computational throughput of what was considered a supercomputer a few decades ago, and today’s compute clusters and supercomputers offer petaflops of performance that was unimaginable in the past. This incredible computing power has been a catalyst in propelling human knowledge to the realm of what used to be science fiction. We can use machine learning models to develop self-diving cars, and recognize image with higher accuracy that humans. We can numerically solve differential and integral equations to predict storm paths with great precision, simulate earthquake waves, climate change, spreading of dangerous diseases, protein folding, and galaxy formations. We can even recreate early moments of our universe immediately after the big bang 13.8 billion years ago.

This incredible computational throughput required to push the boundaries of science is achieved via massive parallelism. Massive parallelism means that today’s supercomputers can have hundreds of thousands of compute nodes that can all execute in parallel. Each compute node has its own local memory, and they are connected to each other via interconnects forming a massively parallel distributed memory cluster. Each node in these clusters can have several multi-core processors that share the local memory. As we step into the future the level of parallelism will continue to grow both at the distributed memory level and shared memory level.
To harness the full potential of these massively parallel systems, applications must have both high single node performance and good scalability. Scalability of an application is its ability to execute with increased performance when the compute resources that it runs on are increased. An application with good scalability will achieve an overall performance that is proportional to the number of parallel compute nodes that it is executed on. Thus, only an application with high single node performance and good scalability can attain a good fraction of the peak computational throughput of the overall system.

Achieving both good scalability and high individual node performance is not an easy task. The major challenge that limits scalability and high performance is the limited data bandwidth. Data bandwidth is the rate at which data can be transferred from its source (data’s initial location) to its destination (where it is required for computation). Depending on the initial location of the data, this bandwidth can be several orders of magnitude slower than the computational throughput. Specifically, in a distributed parallel system, the data required by a compute node can be either located in its local memory, or in the memory of another compute node. The local memory bandwidth can be an order of magnitude slower than peak computation throughput of a compute node, while fetching data from non-local memory can be several orders of magnitude slower. This means that the performance and scalability are bounded by the rate of data movement rather than the computational throughput.

The data bandwidth is further limited in the presence of resource sharing. In a distributed memory system, a single data link may be shared between multiple compute nodes depending on how the nodes are connected. Any data communication
between these nodes must share the same link which reduces the effective data bandwidth per compute node. Similarly, on a single shared memory multi-core node, the local memory bandwidth is shared across multiple cores. In both cases, the effective data bandwidth is reduced due to sharing. To see this consider connecting a water supply pipe to feed ten water taps in parallel. If only one of the ten taps is open then the water will flow out of the tap at the same rate as it flows through the supply pipe. However, if all ten taps are open, then the water will flow out of each tap at one tenth of the rate at which it flows through the supply pipe.

Modern computing systems use caching to address the data bandwidth limitations. Caches are limited fast memory that allows data to be delivered to the compute cores at a significantly faster rate than non-local memory, or local memory. If the same word of data requires multiple accesses then it can be read from slow memory once and stored in fast memory (cache) for subsequent accesses, instead of reading it from slow memory every time. On modern computing systems, there are multiple levels of caches with varying capacity, data bandwidth, latency, and degree of sharing. This creates a complex hierarchical memory structure.

In a parallel cluster, non-local memory is at the top of this memory hierarchy (with the exception of disks when they are available). Next in the hierarchy is the local memory which sits on top of a Level 3 (L3) cache that is shared among all cores in a multi-core processor. Level 2 (L2) and Level 1 (L1) caches are further down the hierarchy. They are private caches, i.e. each core in the multi-core processor has its own L2 and L1 caches. Each core also has a handful of data registers. Data registers are extremely fast data storage that supplies data directly to the compute core. They have enough bandwidth to allow compute cores to run at peak performance.
To achieve good scalability and performance, an application must fully exploit this complex memory hierarchy. The technique for doing so is called locality optimization. The goal of locality optimization is to enhance performance and scalability by rescheduling the computation in a way that increases reuse from the fast memory. This is done by scheduling the computation such that accesses to the same data are closer together in the computation schedule.

During a computation, if the data required resides in the slow memory then it is first transferred to the fast memory (cache) before it can be used for computation. This data will be evicted from the cache once the total volume of unique data transferred between slow memory and the cache exceeds the capacity of the cache. So any reuse that occurs after this eviction will require the data to be moved again from the slow memory. By re-scheduling the computation such that accesses to the same data are closer in the schedule, we increase reuse from cache by avoiding eviction.

This dissertation focuses on developing locality optimization based solutions to enhance scalability and performance of several regular and irregular application domains on massively parallel computing environment as well as on single shared memory multi-core processors. The different application domains explored in this dissertation are distributed Tensor Contractions, Four-Index Transform problem, Convolutional Neural Networks, and Recursive Traversal of k-d Trees. Each of these applications is limited in performance, scalability or both, primarily by data movement costs at a particular level memory hierarchy. To reduce these data movement costs we develop distributed algorithms, lower bound techniques, and compiler and runtime frameworks in this dissertation.
While there is some commonality between the aforementioned application domains, they are mostly orthogonal to each other, and the locality challenges that they face are unique. Due to this, challenges and solutions for each application domain are presented in six self-contained chapters in this dissertation. It is not necessary to read this thesis in a linear fashion. In fact, readers are encouraged to jump directly to the chapter that discusses the application domain that interests them. An introduction to each of the next six chapters is provided below.

A Communication-Optimal Framework for Contracting Distributed Tensors (Chapter 2)

Tensor Contractions are generalized matrix multiplications. They are the most compute intensive operations in a class of methods in Quantum Chemistry called Coupled Clusters. High dimension tensors can be several terabytes in size, and a large tensor contraction can require hundreds of peta flops of computation. Therefore, to finish the computation in a reasonable time, tensor contractions must be scaled to hundreds of thousands of cores. The primary bottleneck in scaling tensor contractions is the communication time across distributed compute nodes.

To overcome this bottleneck, we developed a communication optimal framework for contracting distributed tensors. This framework takes as input the description of the tensor contraction, the size of the input and output tensors, the size and topology of the compute cluster, and the available memory per distributed compute node. Given this input it generates i) a mapping of input and output tensors to the nodes of the compute cluster, and ii) provides a tensor contraction schedule that is guaranteed to minimize the communication volume between compute nodes. We have shown that our framework performs and scales better than the state-of-art on more than 250K
cores.

CAST: Contraction Algorithm for Symmetric Tensors (Chapter 3)

Tensor contractions represent the most compute intensive core kernels in ab initio computational quantum chemistry and nuclear physics. Symmetries in these tensor contractions make them difficult to load balance and scale to large distributed systems. In this chapter, we develop an efficient and scalable algorithm to contract symmetric tensors. We introduce a novel approach that avoids data redistribution during contraction of symmetric tensors while also bypassing redundant storage and maintaining load balance. We present experimental results on two parallel supercomputers for several symmetric contractions that appear in the coupled cluster singles and doubles (CCSD) quantum chemistry method. We also present a novel approach to tensor redistribution that can take advantage of parallel hyperplanes when the initial distribution has replicated dimensions, and use collective broadcast when the final distribution has replicated dimensions, making the algorithm very efficient.

Optimizing the 4-index Transform using Data Movement Lower Bounds Analysis (Chapter 4)

The four-index integral transform is a computationally demanding calculation in many computational chemistry suites such as NWChem. It transforms a four-dimensional tensor from an atomic basis to a molecular basis. This transformation is most efficiently implemented as a sequence of four tensor contractions that each contract a four-dimensional tensor with a two-dimensional transformation matrix. It is very challenging to develop an effective mapping/scheduling of the computation
on a distributed-memory parallel system, minimizing the amount of data movement, across nodes of a parallel machine, as well as within the disk/memory/cache hierarchy. This is because of the huge number of possible choices for loop fusion and tiling, and data/computation distribution across a parallel system, for the sequence of four tensor contractions.

We developed a novel approach based on lower bounds modeling of data movement complexity to address this optimization problem in two distinct scenarios: 1) minimizing data movement in a memory hierarchy, and 2) maximizing the size of tensors that can be transformed within the collective physical memory of a distributed-memory parallel system without use of any disk I/O. This work has resulted in the development of an improved implementation of the four-index transform that improves on the various currently available variants in the NWChem quantum chemistry software suite.

**Optimizing CNNs on Multicores for Scalability, Performance, and Goodput** (Chapter 5)

Convolutional Neural Networks (CNN) are a class of Artificial Neural Networks (ANN) that are highly efficient at the pattern recognition tasks that underlie difficult AI problems in a variety of domains, such as speech recognition, object recognition, and natural language processing. CNNs are, however, computationally intensive to train, and state-of-art implementations on CPUs are sub-optimal.

The general approach to computing CNN on a CPU is to cast it first as a matrix-multiply (GEMM), then use GEMM libraries such as OpenBLAS or MKL to perform
these matrix-multiplications. Converting a convolution into a matrix-multiply requires an unfolding of the input data which can drastically increase the memory footprint of the computation. We present a performance characterization that shows the limitation of this unfold+GEMM approach for the space of convolution networks. Based on this characterization we develop an alternate framework consisting of multiple-code generators that perform convolution directly without converting it into GEMM. Our framework enhances cache and register locality, and reduces the memory footprint, to improve performance and scalability in scenarios when GEMM based approach is not effective.

**On Fusing Recursive Tree Traversals of K-d Trees (Chapter 6)**

Loop fusion is a key program transformation for data locality optimization that is implemented in production compilers. But optimizing compilers for imperative languages currently cannot exploit fusion opportunities across a set of recursive tree traversal computations with producer-consumer relationships. In this thesis, we develop a compile-time approach to dependence characterization and program transformation to enable fusion across recursively specified traversals over k-d trees. We present the FuseT source-to-source code transformation framework to automatically generate fused composite recursive operators from an input program containing a sequence of primitive recursive operators. We use our framework to implement fused operators for MADNESS, Multiresolution Adaptive Numerical Environment for Scientific Simulation. We show that locality optimization through fusion can offer significant performance improvement.
This chapter describes the design and implementation of a layered domain-specific compiler to support MADNESS—Multiresolution Adaptive Numerical Environment for Scientific Simulation. MADNESS is a high-level software environment for the solution of integral and differential equations in many dimensions using adaptive and fast harmonic analysis methods with guaranteed precision. MADNESS uses k-d trees to represent spatial functions and implements operators like addition, multiplication, differentiation, integration on the numerical representation of functions. The MADNESS runtime system provides global namespace support and a task-based execution model including futures.

MADNESS is currently deployed on massively parallel supercomputers and has enabled many science advances. Due to the highly irregular and statically unpredictable structure of the k-d trees representing the spatial functions encountered in MADNESS applications, only purely runtime approaches to optimization have previously been implemented in the MADNESS framework. This chapter describes a layered domain-specific compiler developed to address some performance bottlenecks in MADNESS. The newly developed static compile-time optimizations, in conjunction with the MADNESS runtime support, enable significant performance improvement for the MADNESS framework.
Chapter 2: A Communication-Optimal Framework for Contracting Distributed Tensors

2.1 Introduction

Tensor contractions are higher dimensional analogues of matrix multiplication, and comprise the computationally dominant operations in many-body methods such as Coupled Cluster (CC) [11, 32]. Production computational chemistry software suites such as ACES [38], GAMESS [52], and NWChem [130] partition tensors into blocks, which are distributed among nodes of a distributed memory computer and perform contractions at the block level. Unlike matrix multiplication, for which several efficient distributed-memory parallelization schemes are well known, [118, 87, 65, 132], communication optimized distributed tensor contraction algorithms had not been devised when these codes were developed. Until recently, this was not very critical, as elaborated below using the example of NWChem [130], a widely used parallel quantum chemistry suite.

NWChem implements distributed tensor contractions as follows: each tensor is rearranged into a collection of multidimensional blocks that are spread over the nodes of a distributed memory machine. The Global Arrays (GA) [96] library is used to provide globally addressed storage and convenient one-sided communication operations
for accessing these blocks. Each tensor contraction is implemented as a collection of tasks which retrieve a pair of data blocks from the operand tensors, perform a local contraction of the blocks, and then contribute an additive update to a block of the result tensor. The mapping of tasks to processors is done dynamically using a global lock-based load balancing scheme and non-local data blocks are moved between processors as needed. The location of data blocks in the distributed-memory system is not considered when mapping the local contractions to processors. Since the total number of arithmetic operations from contracting two data blocks is larger than the volume of data moved in getting the data blocks to the processor performing the contraction, such an approach achieved good parallel performance for NWChem on several generations of parallel systems over the last two decades. However, with the number of cores on parallel systems exceeding hundreds of thousands, the communication overhead of distributed tensor contractions now limits the scalability of coupled cluster methods in NWChem on current generation parallel systems [75].

In this chapter, we develop a systematic framework to derive communication efficient algorithms for distributed contraction of arbitrary dimension tensors on distributed memory machines. The framework is comprehensive in automatically modeling potential space–performance trade-offs. For matrix multiplication, a number of distributed algorithms have been developed that explore such space–performance trade-offs, including Canon’s [85] and SUMMA [132] algorithms that use a minimal amount of memory to hold all matrices, and 2.5D and 3D algorithms [120, 5] that utilize extra memory to perform partial replication of matrices to achieve lower communication costs. Similarly, our framework covers a range of algorithms representing space–communication trade-offs for arbitrary dimension tensor contractions.
This chapter makes the following contributions:

- It presents a systematic framework for developing efficient distributed tensor contraction algorithms for arbitrary dimension tensors. This represents an advance towards developing an understanding of parallel tensor contraction algorithms that compares with the comprehensiveness of our understanding of distributed matrix multiplication.

- It develops a cost model to determine the best distributed contraction scheme based on tensor dimensions, processor dimensionality communication parameters, and available memory. Additional memory beyond the minimum required to hold all tensors is automatically exploited to select a higher dimensional scheme with lower communication overhead, analogous to the 2.5D and 3D algorithms known for matrix multiplication.

- It provides a proof of asymptotic communication optimality of the algorithms – for a given tensor contraction to be executed on a parallel machine, the amount of available memory relative to the minimum space to hold the tensors determines an appropriate member in the family of algorithms that asymptotically matches the lower bound of communication for the available memory.

- It presents an experimental evaluation of the framework on a large-scale parallel supercomputer and demonstrates scalability of the framework on up to 256K cores.
2.2 Preliminaries

2.2.1 Definitions and Notations

Tensors are multidimensional arrays. We refer to the space represented by the multidimensional array as the tensor’s data space. A tensor contraction is a binary operation on two tensors and can be represented in general as:

$$C[o_1...k...n] = \sum_{i_1...i_m} A[o_1...k, i_1...m] \times B[i_1...m, o_{k+1...n}]$$ (2.1)

$A$ and $B$ are input tensors that contract to produce tensor $C$. Tensor indices such as $o_1, i_1, i_2$ are labels used to identify orthogonal dimensions in a tensor’s data space and are represented using lowercase letters. It is crucial to remember that by referring to a tensor index, we are strictly referring to the dimension in the tensor’s data space that it represents. For example, in matrix multiplication

$$C[i, j] = \sum_k A[i, k] \times B[k, j]$$ (2.2)

$i, j, k$ are tensor indices, where $i$ represents dimension 0 in $A$ and $C$, $j$ represents dimension 1 in $C$ and $B$, and $k$ represents dimension 1 in $A$ and 0 in $B$ as shown in Fig. 3.1.

Tensor indices serve to match dimensions involved in a contraction. In Exp. 3.1, $o_1, \ldots$ are the external indices, i.e., indices from the input tensors that appear in the result. $i_1, \ldots$ are contraction or internal indices that are common across both inputs and are summed over. Each tensor index appears in exactly two tensors; the external indices appear in an input tensor and the output tensor, while the contraction indices appear in both of the input tensors. Therefore, if we let the dimensionality of $A$, $B$ and $C$ be $d_A$, $d_B$ and $d_C$ respectively, then the total number of unique tensor indices is given by $\frac{d_A + d_B + d_C}{2}$. For example, in matrix multiplication the dimensionality of each
Figure 2.1: Data space and iteration space of matrix multiplication as shown in Eq. 3.2 and Lst. 3.1.

tensor is 2 and the total number of unique indices is 3. This is also the dimensionality of the iteration space of the contraction defined below.

The iteration space of a tensor contraction is a space of all the computation points. It is equivalent to the space of all the loop iterations in the contraction. The dimensionality of the iteration space is equal to the number of loops. For example, matrix multiplication has a 3-dimensional iteration space corresponding to the 3 nested loops for the computation shown in Lst. 3.1. Loop iterators are labels used to identify dimensions of the iteration space and are represented using uppercase letters in this chapter. In a tensor contraction, for each tensor index \( i \), there is a corresponding loop iterator \( I \) such that points along \( I \) in the iteration space accesses data along \( i \) in the data space. Fig. 3.1 shows the loop iterators for matrix multiplication given
Listing 2.1: Matrix Multiplication

1 for i => Loop iterator I
2 for j => Loop iterator J
3 for k => Loop iterator K
4 //contraction expression
5 C[i,j] += A[i,k] x B[k,j]

by Exp. 3.2. \( I, J \) and \( K \) are the loop iterators corresponding to the tensor indices \( i, j \) and \( k \) respectively and corresponds to dimensions 0, 1 and 2 in the iteration space. The loop themselves are shown in Lst. 3.1. A loop iterator may either correspond to an external or contracting tensor index. We call it a external iterator or contracting iterator respectively. \( I \) and \( J \) are external iterators and \( K \) is a contracting iterator.

2.3 Iteration Space and Data Space Mapping

Since the computation described by a contraction needs to happen on a physical processor it is necessary to identify the processor on which each point of the iteration space is computed. Consider a \( l \)-dimensional iteration space and a \( t \)-dimensional torus of processors, which is simply a \( t \)-dimensional mesh of processors with additional links between the first and last processors along each dimension. In this section, we describe different ways to map points in the iteration space to processors in a torus. We do this using an iteration dimension to torus dimension map (\( ipdMap \)) and an iteration point to processor map (\( ippMap \)).

For a loop iterator \( I_k \) in the iteration space, let \( Q_k \) be the number of points along \( I_k \). Similarly, let \( N_j \) be the size of the \( j^{th} \) dimension of the torus. Let \( LI = \{I_1, I_2, ..., I_l\} \) be the set of loop iterators and let \( PD = \{1, 2, 3, ..., t, [\ ]\} \) be the set of processor dimensions of the torus and a null dimension represented by [ ].
Definition 2.3.1. Iteration dimension to processor dimension map ($ipdMap$) is a mapping from $LI$ to $PD$ defined as $ipdMap : IL \rightarrow PD$ such that $\forall A, B \mid A, B \in LI \land ipdMap(A) \neq []$, $ipdMap(A) = ipdMap(B) \iff A = B$.

$ipdMap$ is a mapping from $LI$ to $PD$ such that no two loop iterators can be distributed along the same dimension of the processor grid. We will discuss the implication of this later on in this section. Now consider a set $R_k = \{1, 2, 3, ..., Q_k\}$ which represents the iteration points along loop iterator $I_k$ and a set $P_j = \{1, 2, 3, ..., N_j\}$ which represents processors along the $j^{th}$ dimension of the torus.

Definition 2.3.2. Iteration point to processor map ($ippMap_{kj}$) is a surjective function from $R_k$ to $P_j$, given by $ippMap_{kj} : R_k \rightarrow P_j$. This map exists only if $ipdMap(I_k) = j$.

Points in an $l$-dimensional iteration space can be represented by a $l$-tuple $(i_1, i_2, i_3, ..., i_l)$, where $i_k \in R_k$. Similarly, each processor in a $t$-dimensional torus can be represented by a $t$-tuple, $P[p_1, p_2, ..., p_t]$ where $p_j \in P_j$. Now consider an iteration point given by $IP\langle i'_1, i'_2, i'_3, i'_4, ..., i'_l \rangle$ and let $MP$ be the set of processors to which $IP$ is mapped.

Definition 2.3.3. For a given iteration point $IP$, $MP$ is a set of all the processors that computes $IP$. A processor $P[p'_1, p'_2, p'_3, ..., p'_s] \in MP$ iff $\forall p'_j, p'_j \in MP_j$ where $MP_j \subset P_j$ and is defined as follows:

$$\text{if } \exists I_k \mid I_k \in LI \land ipdMap(I_k) = j \text{ then}$$

$$MP_j = \{ippMap_{kj}(i'_k)\}$$

else

$$MP_j = P_j$$
Table 2.1: Exhaustive list of data space mapping of input tensors

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed Distributed Orthogonal (DDO)</td>
<td>( k_A \rightarrow p_I \land k_B \rightarrow p_J \mid p_I \neq p_J )</td>
</tr>
<tr>
<td>Distributed Distributed Aligned (DDA)</td>
<td>( k_A \rightarrow p_I \land k_B \rightarrow p_J \mid p_I = p_J )</td>
</tr>
<tr>
<td>Distributed Serialized (DS)</td>
<td>( k_A \rightarrow [\ ] \land k_B \rightarrow [\ ] )</td>
</tr>
<tr>
<td>Serialized Serialized (SS)</td>
<td>( k_A \rightarrow [\ ] \land k_B \rightarrow [\ ] )</td>
</tr>
<tr>
<td>Distributed Conflicted (DC)</td>
<td>( e_A \rightarrow p_I \land \exists e_B \mid e_B \rightarrow p_J \land p_I = p_J )</td>
</tr>
<tr>
<td>Distributed Exclusive (DE)</td>
<td>( e_A \rightarrow p_I \land \forall e_B \mid ipdMap(e_B) \neq p_I )</td>
</tr>
<tr>
<td>Serialized (S)</td>
<td>( e_A \rightarrow [\ ] )</td>
</tr>
</tbody>
</table>

The mapping of a tensor’s data space to a processor grid can also be defined in the same way as the iteration space to processor mapping. The iteration space (or data space) to processor mapping has the following properties:

- Each dimension of the iteration space (or tensor’s data space) is either distributed along some dimension of the torus or is serialized. For an iterator \( I_k \), if \( ipdMap(I_k) = j \) then there exists a surjective function from the points in the iteration space along \( I_k \) to processors along dimension \( j \), implying that the iteration points along \( I_k \) are distributed along \( j \). Similarly, if \( ipdMap(I_k) = [\ ] \) then no such mapping exists and hence each point along \( I_k \) will be computed on all processors in the torus. We call this serialization of \( I_k \). In the rest of this proposal we use the notation \( I_k \rightarrow j \) to mean distribution of \( I_k \) along dimension \( j \) of the torus. Similarly we use \( I_k \rightarrow [\ ] \) to mean serialization. Fig. 3.2 shows examples of serialized and distributed dimensions of the iteration space in matrix multiplication.
Figure 2.2: Iteration space mappings for 3-dimensional iteration space of matrix multiplication on to a 3-dimensional torus. In Map1 loop iterators I,J and K are distributed along torus dimensions $p_0$, $p_1$ and $p_2$ respectively. In Map2, loop iterators I and J are distributed along torus dimensions $p_0$ and $p_2$ respectively. Loop iterator K is serialized and the iteration space is replicated along torus dimension $p_1$, which means that each processor along $p_1$ computes the same set of points in the iteration space.

- No two dimensions of the iteration space (or tensor’s data space) can be mapped to the same dimension of the torus. Such mapping has no meaning. An iteration space consists of all points in the product set given by the Cartesian product of all the points in each dimension. It is not possible to form full Cartesian product between two dimensions of the iteration space if they are both mapped to the same dimension of the torus.

- Iteration space (or tensor’s data space) is replicated along those dimensions of the torus where no loop iterator is distributed. This implies redundant computation. An example of this is shown in Fig. 3.2 Map2.
Fig. 3.2 shows the iteration space mapping for matrix multiplication to 3-dimensional torus. These mappings can be written in the notation described above as

\[
Map1 : \{ I \rightarrow p_0, J \rightarrow p_1, K \rightarrow p_2 \} \quad (2.3)
\]

\[
Map2 : \{ I \rightarrow p_0, J \rightarrow p_2, K \rightarrow [] \} \quad (2.4)
\]

2.3.1 Categorization of input tensor distribution

Given a tensor contraction, and a torus, all possible distributions of the input tensors given by the data space mapping can be categorized based on contraction indices and external indices. The exhaustive list is shown in Table 2.1 and each of them are further described below.

Distribution of Contraction Indices

Let \( k \) be a contraction tensor index and let \( k_A \) and \( k_B \) be the occurrence of this index in tensors \( A \) and \( B \) respectively. \( k_A \) and \( k_B \) can each be either serialized or distributed. All different possible distributions of \( k_A, k_B \) are: Distributed, Distributed - Aligned (DDA), Distributed, Distributed - Orthogonal (DDO), Serialized, Distributed (SD) or vise versa, Serialized, Serialized (SS). DDA refers to the case when both \( k_A \) and \( k_B \) are distributed along the same dimension of the torus and DDO refers to distribution along separate dimensions. The precise definition of each of these distributions is given in Table 2.1.

Distribution of External Indices

Let \( e_A \) be an external tensor index in tensor \( A \). \( e_A \) can be either serialized or distributed. We classify distributed external indices into two categories based on other
external indices that may be distributed along the same dimension as $e_A$: Distributed-Conflicting (DC), Distributed-Exclusive (DE). $e_A$ is Distributed-Exclusive if no external index of $B$ is distributed along the same dimension of torus as $e_A$, otherwise it is Distributed-Conflicting (DC).

2.3.2 Problem Statement

Mapping of the iteration space precisely defines where each computation of a tensor contraction occurs, and therefore defines the data that needs to be present in each processor. However, all the data required by the iteration space mapping at a particular processor may not be immediately available depending on the input tensor distribution. In this case, data needs to be communicated to the processor. Given a tensor contraction, we want to find an iteration space mapping, a data space mapping and an algorithm that minimizes the communication cost of contraction for a given amount of memory per processor.

2.4 RRR framework

RRR refers to three communication operators in our framework: Reduction, Recursive Broadcast and Rotation. For a given iteration space mapping, our framework identifies communication directions that are fundamental to the tensor contraction called reuse directions. Using these directions, our framework can compute compatible input and output tensor distributions and systematically generate contraction algorithms for every compatible distribution using the RRR communication operators.
2.4.1 Reuse Directions

Reuse directions are dimensions along the iteration space where tensor elements are reused. Each loop iterator in the iteration space is a reuse direction for exactly one of the participating tensors. In Lst. 3.1, \( i \) is not a tensor index for tensor \( B \) and so \( B \) is reused along the corresponding loop iterator \( I \). Similarly \( j \) and \( k \) are not tensor indices for tensors \( A \) and \( C \) respectively, so the corresponding loop iterators \( J \) and \( K \) are reuse iterators for \( A \) and \( C \) respectively.

**Definition 2.4.1.** Reuse Iterator: In a tensor contraction, if \( I \) is a loop iterator in the iteration space and \( I \) does not have a corresponding tensor index in tensor \( T \), then \( I \) is a reuse iterator for \( T \) in that contraction.

**Definition 2.4.2.** Reuse Dimension: If an iterator \( I \) is a reuse iterator for tensor \( T \) and \( I \rightarrow p_I \), where \( p_I \) is a torus dimension, then \( p_I \) is a reuse dimension for \( T \).

In Fig.2.3 the iteration space and data space for matrix multiplication shown in Lst. 3.1 is mapped to a 2D torus. The iteration space mapping is given by \{ \( I \rightarrow \)
\( p_I, J \rightarrow p_J, K \rightarrow \square \}. \) Since \( I \) and \( J \) are reuse iterators for \( B \) and \( A \) respectively, \( p_I \) is a reuse dimension for \( A \) and \( p_J \) is a reuse dimension for \( B \).

**Definition 2.4.3.** *Fundamental communication algorithm:* An algorithm is a fundamental communication algorithm if each tensor is only communicated along its reuse dimensions during contraction.

Reuse dimensions describe the fundamental data movement directions required by a distributed algorithm. If iterator \( J \) is reuse iterator for \( A \) and \( p_J \) is the corresponding reuse dimension then either \( A \) has to be replicated along \( p_J \) or each element of \( A \) has to be broadcasted along \( p_J \) during the contraction. Our framework only considers fundamental communication algorithms. We show in Sec. 2.6 that for given memory constraints fundamental communication algorithms are communication optimal. Data movement along non-reuse direction is secondary. It implies a misalignment between iteration space mapping and data space mapping. If a data point needs to be communicated but is only used at the destination processor, that simply means that the data was in the wrong place to begin with.

### 2.4.2 Compatible Input Distribution

For a given iteration space mapping, an input distribution is compatible if a fundamental communication algorithm can be constructed to contract the input tensors. We use the following two lemmas to identify such compatible input distributions. If \( i \) is an index of tensor \( T \), then let it be denoted as \( i_T \).

**Lemma 2.4.4.** For a fundamental communication algorithm, if iterator \( I \) is distributed along \( p_I \), i.e., \( I \rightarrow p_I \), and \( I \) is a non-reuse iterator for tensor \( T \) then tensor index \( j_T \mid j_T \neq i_T \) cannot be distributed along \( p_I \).
Table 2.2: Requirements for Compatible Distribution of Input Tensors

<table>
<thead>
<tr>
<th>Iterator Mapping</th>
<th>Compatibility Requirements</th>
<th>Compatible Distributions</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K \to p_r$</td>
<td>$k_A, k_B \to []$</td>
<td>(SS)</td>
<td>EP.2.4.6</td>
</tr>
<tr>
<td></td>
<td>$\lor (k_B \to [] \land k_A \to p_x)$</td>
<td>(DS)</td>
<td>Lem.2.4.4</td>
</tr>
<tr>
<td></td>
<td>$\lor (k_A \to [] \land k_B \to p_x)$</td>
<td>DDA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\lor (k_A \to p_x \land k_B \to p_x)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K \to []$</td>
<td>$k_A, k_B \to []$</td>
<td>SS</td>
<td>Lem.2.4.5</td>
</tr>
<tr>
<td></td>
<td>$\lor (k_B \to [] \land k_A \to p_x)$</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\lor (k_A \to [] \land k_B \to p_x)$</td>
<td>DDO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$p_x \in R_A \land p_y \in R_B$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E \to p_x$</td>
<td>$e_A \to p_x \lor e_A \to []$</td>
<td>(S), DE, DC</td>
<td>EP.2.4.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lem.2.4.4</td>
</tr>
<tr>
<td>$E \to []$</td>
<td>$e_A \to [] \lor e_A \to p_E \mid p_x \in R_A$</td>
<td>S, DE, DC</td>
<td></td>
</tr>
</tbody>
</table>

Proof. Consider an index $j_T \mid j_T \neq i_T$ distributed along $p_I$ i.e., $j_T \to p_I$. Either $J$ is serialized or $J \to p_J \mid p_I \neq p_J$ since $I \neq J$. In case $J \to p_J$, let $[pi, pj]$ represent a node whose coordinates along $p_I$ and $p_J$ are $pi$ and $pj$ respectively. A set of nodes $[pi, *]$, where $*$ represents all nodes along the dimension, requires all elements of $T$ along index $j$ for a given index $i$. Since $j$ is distributed along $p_I$, these elements are held at some set of nodes $[* ,pj']$. Tensor elements need to be communicated from $[* ,pj']$ to $[pi, *]$, i.e., $T$ must be gathered along $p_I$. Similarly if $J$ is serialized, some node along $p_I$ given by $[pi]$ requires all elements held by $[*]$. Again, $T$ has to be gathered along $p_I$, which is a non-reuse direction for $T$. \hfill \square
**Lemma 2.4.5.** For a fundamental communication algorithm, if iterator $I$ is serialized and $I$ is a non-reuse iterator for tensor $T$, then the corresponding index $i_T$ is either serialized or distributed along reuse dimension of $T$.

**Proof.** Serialization of $i_T$ avoids any communication. If $i_T$ is distributed along some dimension $p_J$, then $T$ has to be communicated along $p_J$ since $I$ is serialized. Since only communication along reuse directions is allowed $p_J$ must be reuse direction for $T$. 

For a given iteration space mapping, all input distribution allowed by Lemma 2.4.4 and 2.4.5 are compatible input distributions. These are shown in Table 2.2. The RRR framework uses early pruning to further filter compatible input distribution to avoid unnecessary redundancy in tensor distribution.

**Early Pruning (EP) 2.4.6.** If iterator $I$ is distributed along dimension $p_I$, and if $I$ is a non-reuse iterator for tensor $T$ then index $i_T$ will be distributed along $p_I$ to reduce memory footprint and avoid communication.

EP 2.4.6 is an early pruning strategy in RRR framework. According to Lemma 2.4.4, if $I$ is distributed along $p_I$, then $i_T$ must be either distributed along $p_I$ or $T$ must be replicated along $p_I$ (since no other index of $T$ may be distributed along $p_I$). The former has the tensor index $i_T$ pre-aligned for computation, while the latter has tensor $T$ replicated along a non-reuse dimension. Replicating a tensor along a reuse dimension avoids communication, but replicating a tensor along a non-reuse dimension is wasteful since the replicated data is not used. Furthermore if $i_T$ is distributed along some reuse dimension $p_J \neq p_I$, then communication is required to
along $p_J$ to align $i_T$ with iterator $I$. Therefore, if loop iterator $I \rightarrow p_I$ then $i_T \rightarrow p_I$ in our framework to reduce memory footprint and to avoid unnecessary communication.

For a given mapping of a loop iterator, Table 2.2 shows compatible distributions of the corresponding tensor indices. The distributions inside parenthesis are filtered by EP 2.4.6. In the table, $E$ and $K$ represent external and contraction loop iterators respectively. $p_x$ and $p_y$ are torus dimensions and $R_A$ and $R_B$ are reuse dimensions for tensors $A$ and $B$ respectively. Each dimension of the iteration space corresponds to either a contraction iterator or an external iterator, which is either distributed or serialized. When a contraction iterator $K$ is distributed, the corresponding contraction index in the tensors must be DDA according to EP 2.4.6 along the same dimension as the iterator. If the contraction iterator $K$ is serialized, by Lemma 2.4.5 the corresponding contraction indices must be either SS, DS or DDO where the distributed dimensions are also reuse dimensions. Similarily, if an external iterator $E$ is distributed, then by EP 2.4.6 the corresponding external index must be distributed (DC or DE) along the same dimension as the iterator. Finally if an external iterator $E$ is serialized then by Lemma 2.4.5, the corresponding index must either be serialized (S) or distributed (DC) along a reuse dimension.

2.4.3 Compatible Output Distribution

The location of production of the output tensor is determined solely by the iteration space mapping. EP 2.4.6 implies that indices corresponding to distributed external iterators must be distributed along same dimensions as are the iterators. However, dimensions along which contracting iterators are distributed are reuse dimensions for the output tensor. Indices corresponding to serialized external iterators
maybe mapped along this reuse dimension after their production, or they may be serialized and the entire output tensor maybe replicated along the reuse dimension. Lemma 2.4.5 allows both these distributions.

2.4.4 Fundamental Communication Operators: Reduction, Recursive Broadcast and Rotation

Fundamental Communication Operators are responsible for communicating tensor data along the reuse dimensions. For a given iteration space mapping, a reuse dimension either corresponds to a distributed external loop iterator or a distributed contraction loop iterator. We call them external and internal reuse dimensions respectively. For reasons that will become apparent below, we define Broadcast as communication operator along external reuse dimension and Reduction along internal reuse dimension.
Let $p_x$ and $p_y$ be two dimensions of $p$-dimensional torus and $n_x$ and $n_y$ be the number of processors along $p_x$ and $p_y$. Let $P[x, y]$ represent a general node whose coordinates are $x$ along $p_x$ and $y$ along $p_y$.

- **Broadcast**: If an iterator $I$ is serialized ($I \to [ ]$), but the corresponding index $i_A$ in the input tensor $A$ is distributed along a reuse dimension $p_x$ ($i_A \to p_x$), then $A$ has to be broadcasted along $p_x$ ($p_x$ must be an external reuse dimension since $A$ is an input tensor) for each point along $i_A$ in tensor’s data space. If $I$ is a contraction iterator then, we will call the broadcast operation on $A$ along $p_x$ as **Recursive Broadcast**. The term *recursive* indicates that there can be multiple such contraction indices that are distributed along multiple external reuse dimensions. Similarly if $I$ is an external iterator, then we call the broadcast operation along $p_x$ as **Rotation**. Rotation simply means that $A$ is
rotated along the reuse dimensions, resulting in an efficient pipelined all-to-all broadcast.

- **Reduction**: Distribution of a contracting iterator $K$ along $p_x$ ($K \rightarrow p_x$) implies that the mapping produces partial results along $p_x$ for points in the output tensors data space. The output tensor has to be combined along $p_x$ to obtain the final result. The process of combining partial results of output tensor along internal reuse dimension $p_x$ is called *Reduction*.

For every loop iterator mapping and a compatible tensor index distribution of the corresponding tensor index, Fig. 2.4 shows the communication operator required for communication. Given, any iteration space mapping and a compatible tensor distribution, a fundamental communication algorithm can be constructed for tensor contraction by composition of the RRR operators, using Listing 2.4. Therefore, *Reduction*, *Recursive Broadcast* and *Rotation* are the three fundamental communication operators for tensor contractions. Below we describe these operators in further detail.

**Recursive Broadcast with DDO and DS**

Recursive Broadcast corresponds to serialization of the contracting iterator. Let contraction index $k_A$ and $k_B$ be distributed along processor dimension $p_x$ and $p_y$ respectively. Let $k^m_A$ and $k^m_B$ be range of values of $k_A$ and $k_B$ held at processor $P[m, y]$ and $P[x, m]$ respectively. Notice that except for the diagonal processors $P[m, m]$, the $k_A$ and $k_B$ data are not aligned. In order to contract $k^m_A, k^m_B$ and $k^m_B$ initially held at $P[m, y]$ and $P[x, m]$ need to be held by all nodes. To do this, $P[m, y]$ can broadcast $k^m_A$ along $p_x$ to $P[* , y]$ and $P[x, m]$ can broadcast $k^m_B$ along $p_y$ to $P[x, *]$. Now every processor $P[x, y]$ holds $k^m_A$ and $k^m_B$ and a local contraction $k^m$ can be performed. This
Listing 2.2: Recursive Broadcast

```python
1 c_ortho = DDO contraction indices
2 r_bcast(a, b, c_buf, c_ortho):  
3   k = c_ortho.pop()          
4   A_d, B_d = Dimension of k in A and B
5   for x in range(0, len(k)):  
6     if A_d is not serialized:  
7       if my_rank == x:         
8         BCAST(A[k]) to A_d     
9       else: RECEIVE(A[k])    
10      if B_d is not serialized:  
11        if my_rank == x:       
12          BCAST(B[k]) to B_d   
13        else: RECEIVE(B[k])   
14      if is_empty(c_ortho):   
15        local_dgemm(a[k], b[k], c_buf)  
16      else:                   
17        r_bcast(a[k], b[k],  
18          c_buf, c_ortho)     
```

is done for each $m$ so that entire contraction index $k$ is contracted locally on each node.

This is the SUMMA algorithm for matrix multiplication. The algorithm is given in Lst. 3.2 and is depicted in Fig. 3.3. For a contraction involving multiple contraction index that are DDO, broadcasts are performed recursively along the dimensions of the torus where the contraction indices are distributed. In case only one of $k_A$ or $k_B$ is distributed while the other is serialized (DS), only the tensor corresponding to the distributed contraction index needs to be communicated.

**Reduction with DDA**

*Reduction* corresponds to distribution of the contracting iterator. Let contraction index $k_A$ and $k_B$ be distributed along processor dimension $p_x$. Let $k^m_A$ and $k^m_B$ be the range of values of $k_A$ and $k_B$ held at processor $P[m, y]$. Notice that $k^m_A$ and $k^m_B$ are perfectly aligned on each node, i.e., at node $P[m, y]$, $k^m$ can be contracted since it has both $k^m_A$ and $k^m_B$. After the local contraction each node along $p_x$ i.e., nodes $P[* , y]$ will
hold partial result which can be summed using a Reduction. Notice that Reduction can also be used for contraction indices that are DS or SS. If $k_A$ is serialized and $k_B$ is distributed along $p_x$, then the nodes $P[m, y]$ that hold $k_B^m$ also hold $k_A^m$ since $K_A$ is serialized. This is shown in Fig. 3.3

**Rotation with DC**

Rotation corresponds to serialization of an external iterator. Let external indices $e_A$ and $f_B$ be distributed along $p_x$ of the torus. Hence, $e_A$ and $f_B$ are conflicting. Let $e_A^m$ and $f_B^n$ be ranges of $e_A$ and $f_B$ held at some processor $P[m, y]$. Under such distribution there is no node which holds $e_A^m$ and $f_B^n$ where $m \neq n$. In other words a full Cartesian product between $e_A$ and $e_B$ is not formed.
Since $e_A$ and $f_B$ are external indices that appear in output tensor $C$, these indices cannot be aligned and a full Cartesian product needs to be formed between them. This can be done by rotating either $A$ or $B$ along $p_x$. During each step of rotation of $B$ a processor $P[m, y]$ will receive $f_B^{m-1}$ from $P[m-1, y]$ and will send $f_B^m$ to processor $P[m+1, y]$. Hence, after $n_x$ steps of rotation, $f_B$ will be completely serialized on each node. Refer to Fig. 2.6 and Lst. 2.3.

If the distribution of some contraction index $k$ is SS, then entire range of $k_A$ and $k_B$ is available on each node. No communication is required to contract this index. Similarly, if an external index $e_A$ is serialized, full the Cartesian product is automatically formed between $e_A$ and all other external indices, thus no communication is required w.r.t. $e_A$.

### 2.4.5 Generating Contraction Algorithms

A general contraction algorithm for any given iteration space mapping and a compatible input distribution is given in Lst. 2.4. The algorithm works by first aligning all DDO (and DS) contraction indices corresponding to serialized contraction iterators using *Recursive Broadcast*. Local contractions are performed once the
indices are aligned. Once the DDO (and DS) indices corresponding to serialized contraction iterators are contracted, a reduction is done for indices that are DDA for contraction iterators that are distributed. This completes the algorithm in cases where no external indices corresponding to serialized external iterators are DC, otherwise input tensors with DC external indices corresponding to serialized external iterators are rotated one step at a time. The $RRR$ process is repeated until a full rotation has completed.

### 2.4.6 Dimension Scaling

The dimensionality of a torus grid varies from one machine to another. If this dimensionality is larger than that of the tensors, then the tensors will have to be replicated along those extra dimensions. Furthermore, if the dimensionality of grid is larger than that of the iteration space then redundant computation may be done. While redundant computation is undesirable, the right amount of replication allows full utilization of available memory to lower the communication costs. We can change
the dimensionality of a tensor to avoid replication when unnecessary, and we can change the dimensionality of the torus to control the replication factor.

The dimensionality of a tensor may be decreased using index fusion or increased by index splitting. For instance a single dimension of a tensor $i$ with a range $0 – 15$ may be split into two dimensions $i1$ and $i2$ with range $0 – 3$ and $0 – 3$ respectively. Dimension splitting will increase the dimension of input tensors if a contraction index is split or increase the dimensionality of both input and output tensors if an external index is split. Dimension splitting will also increase the dimensionality of the iteration space therefore avoiding redundant computation or replication of tensors while fully taking advantage of the higher dimensional torus network. An example of such a contraction is 2.5D SUMMA, where a matrix multiplication is performed in a 3D grid by dimension splitting and using a combination of Broadcast and Reduction shown in Fig. 2.6.

The dimensionality of the torus can also be scaled. Consider a $8 \times 8 \times 8$ 3D grid on which matrix multiplication as described in Fig. 2.6 can be performed using 2.5D SUMMA. The torus requires $C$ to be replicated 8 times. Assume that the available memory allows for one $C$ to be replicated only 4 times. In order to benefit from the extra memory, a $16 \times 8 \times 4$ operative torus can be embedded on to $8 \times 8 \times 8$ physical torus, where 2.5D SUMMA can be performed with 4-way replication of $C$. We call this torus scaling.

Torus scaling can cause contention, since the operative torus is different form the underlying physical torus. Hence, it might not always be beneficial. This will be discussed in more detail in Sec. 2.5.3.
2.4.7 Finding optimal mappings

Consider a $d$-dimensional torus. Scaling this torus to a higher dimensional torus results in large amount of contention per link, which will be shown by Lemma 2.5.1. So consider all possible ways of dimension scaling a $d$-dimensional torus to a $d' \leq d$ dimensional torus. For each of these operative tori, consider all possible mapping of the iteration space and all suitable tensor distributions that fits in the memory. Compute the computation cost for each using the cost model and the contention per link and the one with the lowest communication cost. This brute force method finds the communication optimal mapping within our framework. Once this is found, the tensors are distributed accordingly and the contraction is performed. It is possible to reduce the search space based on available memory. In practice, RRR uses this information along with heuristics to reduce the search space.

2.4.8 Initial Tensor Distribution

RRR does not make any assumptions regarding the initial distribution of tensors. Input tensors are distributed systematically based on tensor dimension $m$ and torus dimension $t$. If $m \geq t$, then $i^{th}$ dimension of the tensor is distributed along $i^{th}$ dimension of the torus. If $m < t$, then a $m$-dimensional operational torus is embedded on to the $t$-dimensional physical torus. Then $i^{th}$ dimension of the tensor is distributed along $i^{th}$ dimension of the operational torus.

2.5 Cost Model

Given an operative torus, an iteration space mapping, and a suitable tensor distribution, our cost model predicts the communication cost for the contraction. In
our framework, we avoid redundant computation and the total computation load is balanced. Hence, the goal is to find the optimal communication algorithm.

2.5.1 Cost function based on RRR

Lst. 2.4 gives the general algorithm for any iteration space mapping. The cost function is simply the cost of each RRR component of the general algorithm given by

\[
Total\_\text{cost} = \text{num}\_\text{rotation} \times \left( \text{cost(\text{recursive}\_\text{broadcast} + \text{local}\_\text{compute})} + \text{cost(\text{reduction})} + t_s + t_w \times m \right)
\]

Cost of Rotation

\text{num}\_\text{rotation} is the total number of times an input tensor is rotated due to the presence of a DC external index. If \( p_i \) is the torus dimension along which a DC external index is distributed and \( n_i \) is the number of processors along \( p_i \) then

\[
\text{num}\_\text{rotation} = \Pi n_i.
\]

\( t_s \) and \( t_w \) are the latency and bandwidth of the torus network and \( m \) is the size of the input tensor per node that needs to be rotated.

Cost of Recursive Broadcasts

For a message of size \( m \), the cost of broadcasting it along a torus dimension \( p_i \) with \( n_i \) processors is \( \log(n_i) \times (t_s + t_w \times m) \). Let the size of message per node be \( m_A \) and \( m_B \) for tensors \( A \) and \( B \) respectively. Let \( k \) be a DDO contraction index and let \( k_A \) and \( k_B \) be distributed along \( p_x \) and \( p_y \) respectively. For simplicity, let \( n_x = n_y \).

\[
n_x \times \log(n_x) \times m_A + n_x \times \log(n_x) \times m_B
\]

\[
c \times n_x \times m_A + c \times n_x \times m_B
\]

\[
c \times m_A \times \Pi i n_i + c m_B \times \Pi j n_j
\]
Eq. 2.5 gives the total communication cost for broadcasting. The broadcast is pipelined in our implementation that gets rid of the log term and results in Eq. 2.6 where $c$ is some constant dependent on blocking of the contraction indices. For interested readers we refer to the SUMMA paper by Geijn et al. [132]. For tensor contractions with multiple DDO contraction indices, each distributed along $p_i$ and $p_j$, the communication cost can be computed Eq. 2.7.

**Cost of Reduction**

The cost of reduction has been studied rigorously and several algorithms exist. If data of size $m$ need to be reduced among $n$ processors, the cost is of a non-pipelined algorithm is $\log(n)(t_s + t_w \times m)$, where $t_s$ and $t_w$ are latency and bandwidth of the torus. As with broadcasts, reductions can also be pipelined to remove the log term.

### 2.5.2 Memory required for a contraction

Each dimension of a tensor is either distributed or serialized. Let a tensor index $i \rightarrow p_i$. Let size of $p_i$ be $n_i$. If the total size of the tensor is $S$, then the memory required for this tensor per node is $\frac{S}{\prod n_i}$. Memory required for a contraction per node is the sum of memory required by input and output tensors. Additionally, memory will be required for send and receive buffers depending on the implementation.

### 2.5.3 Contention with Torus Scaling

In the RRR framework, the communication is pipelined, i.e., when a broadcast is done along a processor dimension $p_x$, each node receives data from the previous processor along $p_x$ and forwards it to the next processor along $p_x$. Here, we establish
an upper bound on the contention per link along $p_x$ by examining a situation where every node along $p_x$ is sending data to every $(i + 1)^{th}$ node along $p_x$.

Consider $n_x \times n_y$ sub-torus, where $n_x$ and $n_y$ are number of nodes along $p_x$ and $p_y$. Consider an operational torus $cn_x \times \frac{n_y}{c}$, where $c$ is a scaling constant.

**Lemma 2.5.1.** If every $i^{th}$ node along $p_x$ sends a message to every $\left((i + 1) \% cn_x\right)^{th}$ node along $p_x$ then the upper bound on the contention per link is given by $O\left(1 + 2\frac{n_y/c-1}{n_x}\right)$.

**Proof.** Consider the $cn_x \times \frac{n_y}{c}$ operational torus. This may be partitioned into $c$, $n_x \times \frac{n_y}{c}$ meshes. Let $M1$, $M2$ and $M3$ be three such adjacent meshes. The number of links connecting $M1$ with $M2$ in the operative mesh is $\frac{n_y}{c}$. In the underlying torus, the number of actual physical links connecting $M1$ and $M2$ is $n_x$. Therefore, the contention is given by $\frac{n_y}{cn_x}$ if all the $n_x^{th}$ node along $p_x$ in $M1$ are sending messages to the $1^{st}$ nodes along $p_x$ in $M2$. Of the $\frac{n_y}{c}$ messages sent, $\frac{n_y}{c} - 1$ have a path that passes internally, i.e. there is no direct link between sender and receiver, and so the message has to be routed though an alternate path that passes through the mesh.

In the operative torus, $i^{th}$ node along $p_x$ sends a message to every $\left((i + 1) \% cn_x\right)^{th}$ node along $p_x$. These message paths conflicts with message received from $M1$ at $M2$ and messages sent from $M2$ to $M3$ that gets routed internally. There are $\frac{n_y}{c} - 1$ messages and $n_x$ path, the contention is given by $\frac{n_y/c-1}{n_x}$. Since we are computing an upper bound, let us assume that messages sent internally conflicts with message received internally. Therefore, the contention add up and the upper bound on the contention per link is given by $1 + 2\frac{n_y/c-1}{n_x}$. 

The operative torus has a link between first and the last nodes along $p_y$, which is absent in the underlying physical torus. This suggests that contention per link along
\( p_y \) is simply 2. These contention factors are taken into account while computing cost for a given iteration space mapping and a suitable tensor distribution on the operative torus.

### 2.6 Optimal Communication in RRR

Our approach to proving optimality is similar to that of Irony et al. [65] in their proof of optimality for 2D, 2.5D, and 3D algorithms for distributed matrix multiplication. They show that for a given amount of memory \( M \) and \( W \) computations per node, the total volume of data that must flow through a node has the lower bound

\[
\Omega \left( \frac{W}{M^{\frac{1}{2}}} \right)
\]

This communication lower bound also holds for tensor contractions. A tensor contraction can be expressed as an equivalent matrix multiplication by fusing all contraction indices into a single contraction index, and all external indices from each of the input tensors into two external indices; one for each of the input tensors. The reduced 3D iteration space obtained from index fusion corresponds to standard matrix multiplication, thus the lower bound also holds for arbitrary tensor contractions.

To prove that RRR framework is communication optimal, we show that for a given amount of memory, an algorithm can be constructed within the framework that achieves the lower bound.

**Lemma 2.6.1.** Consider a \( d \)-dimensional torus with \( p \) nodes on each dimension. For a nonnegative \( r < d \), \( d - r = 2i \ | \ i \in \mathbb{N} \), and \( \alpha \leq d - r \ | \ \alpha \in \mathbb{N} \), an algorithm can be constructed in the RRR framework for any contraction \( C = A.B \), with a communication bound of \( O\left( \frac{|A|}{p^d} \times p^{\frac{d - r - \alpha}{2}} + \frac{|B|}{p^d} \times p^{\frac{d - r + \alpha}{2}} \right) \) and memory space requirement \( O\left( \frac{|A|}{p^d} + \frac{|B|}{p^d} + \frac{|C|}{p^{d-r}} \right) \).
Proof. Let \( E_A \) and \( E_B \) be the number of external indices and \( K \) be the number of contracting indices. Using index splitting, let \( E'_A \geq \frac{d-r+\alpha}{2} \) and \( E'_B \geq \frac{d-r-\alpha}{2} \) where \( E'_A \) and \( E'_B \) are number of operative external indices of \( A \) and \( B \) respectively that are scaled to satisfy the constraints. Similarly, let \( K' > r \) using index splitting.

Now consider an iteration space mapping where \( r \) contracting iterators, \( \frac{d-r-\alpha}{2} \) external iterators from \( A \) and \( \frac{d-r+\alpha}{2} \) external iterators from \( B \) are distributed along \( d \) dimensions of the torus and the remaining iterators are serialized. Lemma 2.4.5 allows a distribution of input tensors, where \( r \) contracting indices are DDA, \( \frac{d-r+\alpha}{2} \) external indices from \( A \) and \( \frac{d-r-\alpha}{2} \) external indices from \( B \) are DE, \( \frac{d-r-\alpha}{2} \) contracting indices are DDO, and \( \alpha \) contracting indices are DS with distributed contracting indices from tensor \( B \). Lst. 2.4 and Fig. 2.4 show that the contraction can be performed using Recursive Broadcast and Reduction. Eq. 2.7 gives the communication cost for Recursive Broadcast for this algorithm. \( A \) and \( B \) are not replicated, so \( m_A \) and \( m_B \) in Eq. 2.7 are \( \frac{|A|}{p^d} \) and \( \frac{|B|}{p^d} \) respectively, and \( \Pi_i n_i = p^{\frac{d-r-\alpha}{2}} \) and \( \Pi_j n_j = p^{\frac{d-r+\alpha}{2}} \).

Hence, the communication cost is \( O\left( \frac{|A|}{p^d} \times p^{\frac{d-r-\alpha}{2}} + \frac{|B|}{p^d} \times p^{\frac{d-r+\alpha}{2}} \right) \). \( C \) is replicated along \( r \) dimensions in this distribution, hence \( m_C \) is \( \frac{|C|}{p^{d-r}} \). Therefore, the total memory is \( O(m_A + m_B + m_C) = O\left( \frac{|A|}{p^d} + \frac{|B|}{p^d} + \frac{|C|}{p^{d-r}} \right) \).

Lemma 2.6.2. Consider a \( d \)-dimensional torus with \( p \) nodes on each dimension. For a nonnegative \( r < d \) and \( d - r = 2i \mid i \in \mathbb{N} \), an algorithm can be constructed in the RRR framework for any contraction \( C = A.B \), which requires no communication of the input tensors and occupies \( O\left( \frac{|A|}{p^d} \times p^{\frac{d-r}{2}} + \frac{|B|}{p^d} \times p^{\frac{d-r}{2}} + \frac{|C|}{p^{d-r}} \right) \) memory.

Proof. Consider the same notations and iteration space mapping described in Lemma 2.6.1 with \( \alpha = 0 \). EP 2.4.6 and Lemma 2.4.5 allow a distribution of input tensors, where \( r \) contracting indices are DDA, \( \frac{d-r}{2} \) external indices from \( A \) and \( B \) each are
DE, and all the remaining indices are serialized. Lst. 2.4 and Fig. 2.4 show that the contraction can be performed using just Reduction. Therefore, no communication of the input tensors is required for contraction. However, notice that $A$ and $B$ are replicated along the reuse dimensions. Since each input tensor has $\frac{d-r}{2}$ reuse dimensions, $m_A$ and $m_B$ are $\frac{|A|}{p^d} \times p^{\frac{d-r}{2}}$ and $\frac{|B|}{p^d} \times p^{\frac{d-r}{2}}$ respectively, and $C$ is replicated along $r$ dimensions in this distribution; hence $m_C$ is $\frac{|C|}{p^{d-r}}$. Therefore, the total memory is $O\left(\frac{|A|}{p^d} \times p^{\frac{d-r}{2}} + \frac{|B|}{p^d} \times p^{\frac{d-r}{2}} + \frac{|C|}{p^{d-r}}\right)$.

Using Lemma 2.6.1 and Lemma 2.6.2 we now show that it is possible to obtain an algorithm for a tensor contraction that achieves the communication lower bound of Irony et al.

**Lemma 2.6.3.** Consider a $d$-dimensional torus with $p$ nodes on each dimension. If the amount of memory per node is $M$, then for $|A| = p^\alpha |B|$ where $\alpha \geq 1$, it is always possible to construct an algorithm within RRR for $C = A.B$ that achieves Irony et al.’s lower bound of $\Omega \left( \frac{W}{M^{\frac{3}{2}}} \right)$ for communication, where $W$ is the minimum amount of computation per node.

**Proof.** For any contraction $C = A.B$, the total number of multiplications is given by $\left(\frac{|A||B||C|}{p^d}\right)^{\frac{1}{2}}$. Since there are $p^d$ distributed nodes, at least one of them must do $\frac{\left(\frac{|A||B||C|}{p^d}\right)^{\frac{1}{2}}}{p^d}$ amount of work. This will be our $W$. The amount of memory available can written as a function of the largest tensor. Since $|A| \geq |B|$, we have two cases: either $|C| \geq |A|$ or $|C| < |A|$. We solve for each case. Let $r$, $r'$ and $f$ be nonnegative integers.

**CASE I:** Let $|C| \geq |A|$ and $M = O\left(\frac{|C|}{p^{d-r}}\right)$. Substituting for $M$ and $W$ in Irony et al.’s lower bound expression gives us $\Omega \left( \frac{|A|}{p^d} \times p^{\frac{d-r-\alpha}{2}} \right)$. Since $|C| \geq |A|$, Lemma 2.6.1
shows that there is an algorithm within RRR that achieves this communication for the given \( M \). This strictly gives the amount of data volume that needs to pass through at least one node during the contraction. Notice that if \( r \geq \frac{d}{3} \), then \( \Omega \left( \frac{|A|}{p^d} \times p^{\frac{d-r}{2}} \right) \leq M \), i.e., contraction must be possible without any input communication. The memory term in Lemma 2.6.2 reduces to \( M \) for \( r \geq \frac{d}{3} \). Hence, it shows that such an algorithm can be constructed within RRR.

CASE II: Let \( |C| < |A| \) and \( M = O \left( \frac{|A|}{p^{d-r}} \right) \). Also, let \( |A| = p^r |C| \). Substituting for \( M \) and \( W \) in Irony et al.’s lower bound expression gives us

\[
\Omega \left( \frac{|A||C|^{\frac{1}{2}}}{p^{\alpha/2}p^{\frac{d-r}{2}}} \times p^{\frac{d-r'}{2}} \right) = \Omega \left( \frac{p^{r'}}{p^{\alpha/2}p^{\frac{d-r}{2}}} \right)
\]

\[
= \Omega \left( \frac{|A|}{p^{\alpha/2}p^{\frac{d-r'}{2}}} \right)
\]

(2.8)

By substituting \( r = r' + f \), Lemma 2.6.1 shows that there is an algorithm within RRR that achieves this communication bound for the given \( M \). Notice that if \( r' \geq \frac{d-f}{3} \), then \( \Omega \left( \frac{|A|}{p^{\alpha/2}p^{\frac{d-f}{2}}} \right) \leq M \), i.e., contraction must be possible without any input communication. If \( r \) is substituted with \( r' + f \) for \( r' = \frac{d-f}{3} \) in Lemma 2.6.2, the memory term reduces to to \( M \). Hence, Lemma 2.6.2 shows that such an algorithm can be constructed within RRR.

The proof shows that RRR is communication optimal for any arbitrary tensor contraction, where \( |A| = p^\alpha |B| \) and \( \alpha > 1 \). When \( B \geq A \), a similar proof can be constructed by substituting \( A \) with \( B \). Therefore our proof of optimality is fully general. Notice that the proof assumes a perfectly square torus of size \( p \) on each dimension and allows replication of tensors by factors of form \( p^r \) only. These are not
restrictions of the framework, but rather simplifications used in attempt to keep the proof simpler and shorter.

2.7 Experimental Evaluation of RRR

In this section, we present experimental results evaluating distributed algorithms developed using the RRR framework. We perform comparisons with Cyclops Tensor Framework (CTF) v1.1 [121] and NWChem Proxy. We also perform a cost model verification experiment for the RRR framework.

NWChem Proxy is an application that models the computation, communication, and dynamic load-balancing pattern used in the NWChem TCE module [13, 130]. All tensors are decomposed into tiles, where each contraction is broken into get-compute-accumulate tasks on these tiles. The application utilizes a lock-based dynamic load-balancing scheme. Similar proxy applications for modeling NWChem TCE can be found in [80, 98]. The proxy application is implemented using Global Arrays (GA) 5.2 [95] and ARMCI-MPI, an implementation of ARMCI one-sided communication interface using MPI RMA [39].

The experiments for comparison with CTF and NWChem were performed on IBM Blue Gene/Q Mira at Argonne. The Mira system consists of 48 racks, 1024 nodes per rack, each node with 1.6GHz 16-way core processor and 16 GB memory, and a 5D torus. The implementation of our framework is written in C/C++ and utilizes MPI for communication and BLAS for computation. Computational expensive routines are threaded using OpenMP.
2.7.1 Scalability of RRR

Scalability results are shown for five different contractions chosen to reflect variations in sizes of tensors and arithmetic intensity:

1. $C[i,j,m,n] = A[i,j,k,l] \times B[k,l,m,n]$
2. $C[h,i,j,m,n,o] = A[h,i,j,k] \times B[k,m,n,o]$
3. $C[i,m] = A[i,k,l,h] \times B[m,k,l,h]$
4. $C[i,j,m,n] = A[i,j,m,k] \times B[k,n]$
5. $C[i,j,m] = A[i,j,k,l] \times B[k,l,m]$

For each of the tensor contractions above, RRR can produce multiple contraction algorithms based on different iteration space mappings and compatible input distributions. The algorithms for this experiment are the ones with lowest communication cost, chosen based on the cost model. No assumptions about the initial distribution of tensors were made (refer to Sec. 2.4.8) and the redistribution time to distribute input tensors into compatible distribution required by the algorithm was included in the time reported. Fig. 3.10 shows that both RRR and CTF have good strong scalability on BG/Q. For Contraction 1 and 5 the absolute performance for both RRR and CTF are similar. RRR is faster than CTF on Contractions 2 and 3, while CTF is faster on Contraction 4. The communication time in contraction 4 is very small relative to the total time for RRR. However, local computation such as packing, unpacking, and local transpose required for this contraction dominates the total time – efficient implementation of these local operations (orthogonal to the main developments in this chapter) have not been addressed in the current implementation.
Figure 2.7: Scalability of contractions 1, 3, 4 and 5 with $N = 384$ and contraction 2 with $N = 64$ on up to 16,384 nodes (262,144 cores) on BG/Q.
2.7.2 Comparison with NWChem Proxy

Table 2.3: Scalability of Contraction 1 for N=256 on BG/Q in Seconds

<table>
<thead>
<tr>
<th>Cores</th>
<th>NWChem Proxy</th>
<th>RRR(No Replication)</th>
<th>RRR(With Replication)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>36.89</td>
<td>32.63</td>
<td>20.43</td>
</tr>
<tr>
<td>16384</td>
<td>19.94</td>
<td>13.07</td>
<td>6.63</td>
</tr>
<tr>
<td>65536</td>
<td>9.07</td>
<td>4.87</td>
<td>2.023</td>
</tr>
<tr>
<td>262144</td>
<td>9.22</td>
<td>2.63</td>
<td>0.84</td>
</tr>
</tbody>
</table>

Table 2.3 shows scalability of RRR and NWChem Proxy for Contraction 1. Column 3 and 4 shows contraction time for two algorithms generated by RRR. The former does not take advantage of extra available memory while the latter does. From column 2 and 3, it can be seen that NWChem Proxy and RRR have similar contraction times on 4096 cores, but RRR has better strong scalability as the cores increase. The communication-avoiding algorithm in column 4 is faster and scales better. Both RRR and NWChem stores tensors as tiles, this makes RRR a viable replacement for the Tensor Contraction Engine, which is the ultimate goal for developing the RRR framework.
2.7.3 Cost Model Verification

Table 2.4: Iteration Space Map and Data Space Maps

<table>
<thead>
<tr>
<th></th>
<th>Map1</th>
<th>Map2</th>
<th>Map3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipdMap</td>
<td>{I \rightarrow p_1, J \rightarrow p_2,</td>
<td>{I \rightarrow p_1, J \rightarrow p_2,</td>
<td>{I \rightarrow p_1, J \rightarrow p_2,</td>
</tr>
<tr>
<td></td>
<td>A \rightarrow [], B \rightarrow [], K \rightarrow p_3}</td>
<td>A \rightarrow p_3, B \rightarrow [], K \rightarrow []}</td>
<td>A \rightarrow p_3, B \rightarrow [], K \rightarrow []}</td>
</tr>
<tr>
<td>A</td>
<td>{i \rightarrow p_1, j \rightarrow p_2, k \rightarrow p_3}</td>
<td>{i \rightarrow p_1, j \rightarrow p_2, k \rightarrow p_3}</td>
<td>{i \rightarrow p_1, j \rightarrow p_2, k \rightarrow []}</td>
</tr>
<tr>
<td>B</td>
<td>{a \rightarrow p_1, b \rightarrow p_2, k \rightarrow p_3}</td>
<td>{a \rightarrow p_3, b \rightarrow [], k \rightarrow p_2}</td>
<td>{a \rightarrow p_3, b \rightarrow [], k \rightarrow []}</td>
</tr>
<tr>
<td>C</td>
<td>{i \rightarrow p_1, j \rightarrow p_2,</td>
<td>{i \rightarrow p_1, j \rightarrow p_2,</td>
<td>{i \rightarrow p_1, j \rightarrow p_2,</td>
</tr>
<tr>
<td></td>
<td>a \rightarrow p_3, b \rightarrow []}</td>
<td>a \rightarrow p_3, b \rightarrow []}</td>
<td>a \rightarrow p_3, b \rightarrow []}</td>
</tr>
</tbody>
</table>

The Cost model verification was done on 256 nodes of BG/Q VESTA and 256 nodes of Cray XE6 Hopper. VESTA is architecturally similar to MIRA but only has two rack of 1024 nodes. The Cray XE6 supercomputer consists of dual socket 12core AMD MagnyCours 2.1GHz and 32GB memory per node. Each core has its own L1 (64KB) and L2 (512KB) caches, one L3 (6MB) cache shared between 6 cores. Nodes are connected through a 3D torus Cray Gemini network. For the experiment a 4x8x8 operative torus was embedded on to the 256 nodes. The following contraction was

Table 2.5: Predicted time complexities for different mappings

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Memory required</th>
<th>Communication Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map1</td>
<td>(m_A + m_B + m_C)</td>
<td>(2 \times N_3 (m_C) + N_1 \times N_2 (m_C + m_B))</td>
</tr>
<tr>
<td>Map2</td>
<td>(m_A + N_3 \times m_B + m_C)</td>
<td>((3 \times N_3)m_A + (3 \times N_2)m_B \times p)</td>
</tr>
<tr>
<td>Map3</td>
<td>(N_3 \times m_A + N_1 \times N_2 \times m_B + m_C)</td>
<td>0</td>
</tr>
</tbody>
</table>
computed for varying index-dimension mapping and data space mappings.

\[ C[i, j, a, b] = A[i, j, k] \times B[k, a, b] \]  \hspace{1cm} (2.9)

Figure 2.8: a) Actual vs Predicted on 256 BG/Q nodes. b) Actual vs Predicted on 256 Cray XE6 nodes.

Table 2.4 shows the iteration space mapping and data space mappings on to a 3D torus with \( p_1, p_2 \) and \( p_3 \) as its dimensions and \( N_1 = 4 \) and \( N_2, N_3 = 8 \). \( Map_1 \) performs rotation and reduction, \( Map_2 \) uses the broadcast and \( Map_3 \) does not need any communication. Contractions were run for perfectly square tensors of sizes 256, 384 and 512. Since there is no replication of the iteration space in any of the Maps, the iteration space is equally divided among all processors in the torus. Hence, the computation cost is the same for all processors. Table 2.5 shows the memory required by each of the Maps and the communication cost based on the cost model. \( m_A, m_B \) and \( m_C \) are \( \frac{|A|}{256}, \frac{|B|}{256} \), and \( \frac{|C|}{256} \) respectively. In Fig. 2.8, M1, M2 and M3 represent \( Map_1, Map_2 \) and \( Map_3 \). The cost model ranks M1>M2>M3 as the order for running time. The predicted and the actual values agree.
Chapter 3: CAST : Contraction Algorithm for Symmetric Tensors

3.1 Introduction

Tensor contractions are higher-dimensional analogues of matrix-matrix products, and comprise the computationally dominant operations in most many-body methods in computational physics and chemistry, such as coupled cluster methods [11, 32]. Production parallel computational chemistry software, such as ACES [38], GAMESS [52], and NWChem [130] partitions tensors into blocks that are distributed among nodes in a distributed-memory parallel computer to exploit parallelism in performing tensor contractions. However, none optimize for inter-processor communication in performing distributed tensor contractions. Unlike the case for matrix-matrix multiplication, for which several efficient distributed-memory parallelization schemes have been developed [118, 87, 65, 132], representing different space-time trade-offs, no communication-optimized, distributed-contraction algorithms were known for tensors when these parallel quantum chemistry codes were initially developed.

A significant complication in developing effective distributed tensor contraction algorithms is that the tensors used in computational models in quantum physics and
chemistry exhibit symmetry over multiple dimensions. Exploitation of the symmetry is critical, both to save storage and to avoid unnecessary arithmetic operations. Therefore, all existing parallel quantum chemistry software represents the distinct elements of symmetric tensors in blocks distributed uniformly among nodes of a parallel system and dynamically performs the required movement of blocks to the nodes where block-block contractions are performed. This strategy was satisfactory in practice until recent times. However, with the number of cores on parallel systems exceeding hundreds of thousands, the communication overhead of distributed tensor contractions now limits the scalability of coupled cluster methods [75].

Although some studies over the last two decades have investigated optimizing communication of distributed tensor contractions [24, 48], with the exception of the recently developed Cyclops Tensor Framework (CTF) [118], a significant limitation of prior efforts is that symmetry in tensors was not exploited and therefore they are not practically useful for quantum chemistry codes.

In this chapter, we develop an efficient Contraction Algorithm for distributed Symmetric Tensors (CAST). CAST is built on top of a systematic framework to represent a family of distributed tensor contraction schemes-the RRR framework [107, 105]. The RRR framework can choose an input distribution of tensors that minimizes the communication cost. But the best distribution may involve replication along certain dimensions of the physical grid, especially when different tensors have different dimensionality. To get the input tensors into the best distribution for different contractions, redistribution is required. In this chapter, we also present a novel tensor redistribution algorithm that minimizes link contention and maximizes concurrency in data transfers when the initial and/or final distribution features replicated dimensions.
This chapter makes the following contributions:

- It develops a scalable and load-balanced approach to distributed contraction of symmetric tensors. The scheme avoids the need for explicit transpose or redistribution of tensors during contraction. Further, it uses a block-cyclic distribution of tensors, compatible with the representation used in existing parallel quantum chemistry software suites. To the best of our knowledge, this is the first efficient algorithm for symmetric tensor contractions that demonstrates all of these properties.

- It presents a novel approach for efficient redistribution of tensors in a multi-dimensional torus network when the source and/or target distributions involve replication along one or more dimensions of the processor grid.

- It provides an experimental evaluation on two large-scale parallel supercomputers using a number of representative tensor contractions from many-body methods in quantum chemistry, demonstrating lower inter-processor communication cost than the current state of the art.

- It demonstrates scalability of the algorithm on up to 256 K cores.

The rest of the chapter is organized as follows. In section II, we define important concepts related to tensor contractions. Section 3.2, reviews aspects of the RRR framework [105, 107] that are relevant to this work. In Section 3.3, we develop the Contraction Algorithm for Symmetric Tensors (CAST). In Section 3.5, we present our redistribution algorithm. Experimental results are presented in Section 3.6, and related work is presented in Section 5.2. Section VIII concludes the chapter.
Figure 3.1: Data space and iteration space of matrix multiplication as shown in Eq. (3.2) and Lst. 3.1.

Listing 3.1: Matrix Multiplication

```plaintext
1 for i => Loop iterator I
2 for j => Loop iterator J
3 for k => Loop iterator K
4 //contraction expression
5 C[i, j] += A[i, k] \times B[k, j]
```

3.2 Definitions

Tensors are multidimensional arrays. We refer to the space represented by the multidimensional array as the tensor’s data space. A tensor contraction is a binary operation on two tensors and can be represented in general as:

\[
C[o_1...k...n] = \sum_{i_1...i_m} A[o_1...k, i_1...m] \times B[i_1...m, o_{k+1...n}] \tag{3.1}
\]
A and B are input tensors that contract to produce tensor C. Tensor indices, such as $o_1, i_1, i_2$, are labels used to identify orthogonal dimensions in a tensor’s data space and are represented using lowercase letters. It is crucial to remember that by referring to a tensor index, we are strictly referring to the dimension in the tensor’s data space that it represents. For example, in matrix multiplication

$$C[i, j] = \sum_k A[i, k] \times B[k, j]$$  \hspace{1cm} (3.2)

$i, j, k$ are tensor indices, where $i$ represents dimension 0 in $A$ and $C$, $j$ represents dimension 1 in $C$ and $B$, and $k$ represents dimension 1 in $A$ and 0 in $B$ as shown in Fig. 3.1. Tensor indices serve to match dimensions involved in a contraction. In Exp. 3.1, $o_1, \ldots$ are the external indices, i.e., indices from the input tensors that appear in the result. $i_1, \ldots$ are contraction or internal indices that are common across both inputs and are summed over.

The iteration space of a tensor contraction is a space of all the computation points. It is equivalent to the space of all the loop iterations in the contraction. Loop iterators are labels used to identify dimensions of the iteration space and are represented using uppercase letters in this chapter. In a tensor contraction, for each tensor index $i$, there is a corresponding loop iterator $I$ such that points along $I$ in the iteration space access data along $i$ in the data space. Fig. 3.1 shows the loop iterators for matrix multiplication given by Exp. 3.2. $I, J,$ and $K$ are the loop iterators corresponding to the tensor indices $i, j,$ and $k$ respectively and correspond to dimensions 0, 1, and 2 in the iteration space. The loops themselves are shown in Lst. 3.1. A loop iterator may either correspond to an external or contracting tensor index. We call it a external iterator or contracting iterator, respectively. $I$ and $J$ are external iterators, and $K$ is a contracting iterator.
3.3 Review of the RRR Framework

CAST is built on top of the RRR framework [105, 107]. In this section, we review the aspects of the RRR framework that are essential background for developing CAST. For readers interested in further understanding the RRR framework, please refer to [105, 107].

3.3.1 Iteration Space and Data Space Mapping

Because the computation described by a contraction needs to happen on a physical processor, it is necessary to identify the processor on which each point of the iteration space
space is computed. In this section, we describe different ways to map points in the iteration space (or tensor’s data space) to processors in a torus.

- Each dimension of the iteration space (or tensor’s data space) is either distributed along some dimension of the torus or is serialized. If an iterator $I_k$ is distributed along processor dimension $j$, then a surjective function exists from the points in the iteration space along $I_k$ to processors along dimension $j$, implying that the iteration points along $I_k$ are distributed along $j$. Conversely, if $I_k$ is serialized, no such mapping exists. Hence, each point along $I_k$ will be computed on all processors in the torus. In the rest of the chapter, we use the notation $I_k \rightarrow j$ to mean distribution of $I_k$ along dimension $j$ of the torus. Similarly, we use $I_k \rightarrow [ ]$ to mean serialization. Fig. 3.2 shows examples of serialized and distributed dimensions of the iteration space in matrix multiplication.

- No two dimensions of the iteration space (or tensor’s data space) can be mapped to the same dimension of the torus.

- Iteration space (or tensor’s data space) is replicated along dimensions of the torus where no loop iterator is distributed. This implies redundant computation. An example of this is shown in Fig. 3.2 Map2.

Fig. 3.2 shows the iteration space mapping for matrix multiplication to three-dimensional torus. These mappings can be written in the described notation above as:

$$Map1 : \{I \rightarrow p_0, J \rightarrow p_1, K \rightarrow p_2\}$$

$$Map2 : \{I \rightarrow p_0, J \rightarrow p_2, K \rightarrow [ ]\}$$
Figure 3.3: (a) $C[i,j] = A[i,k].B[k,j]$ DDO Matrix Multiplication. The double arrows along $p_x$ and $p_y$ show the dimension of the broadcast for $k_A$ and $k_B$, respectively.

Listing 3.2: Recursive SUMMA

```python
1 c_ortho = DDO contraction indices
2 r_summa(a, b, c_buf, c_orthogonal):
3   k = c_ortho.pop()
4   A_d, B_d = Dimension of k in A and B
5   for x in range(0, len(k)):
6     if A_d is not serialized:
9       if my_rank == x:
10          BCAST(A[k]) to A_d
13       else: RECEIVE(A[k])
16     if B_d is not serialized:
19       if my_rank == x:
22         BCAST(B[k]) to B_d
25       else: RECEIVE(B[k])
28     if is_empty(c_ortho):
31       local_dgemm(a[k], b[k], c_buf)
34     else:
37       r_summa(a[k], b[k],
39         c_buf, c_ortho)
```

55
3.3.2 Recursive Broadcast with DDO

CAST uses a data communication operator in RRR called recursive broadcast. Recursive broadcast is used when the contracting iterator \( K \) is serialized, but the corresponding contracting indices \( k_A \) and \( k_B \) are distributed along different dimensions of the torus (\( K \rightarrow [ ] \land k_A \rightarrow p_{k_A} \land k_B \rightarrow p_{k_B} \land p_{k_A} \neq p_{k_B} \)). In RRR, this distribution is called distributed distributed orthogonal (DDO). Recursive broadcast works as follows:

Let contraction indices \( k_A \) and \( k_B \) be distributed along processor dimensions \( p_x \) and \( p_y \), respectively. Let \( k^m_A \) and \( k^m_B \) be ranges of values of \( k_A \) and \( k_B \) held at processors \( P[m, y] \) and \( P[x, m] \), respectively. Except for at the diagonal processors \( P[m, m] \), the \( k_A \) and \( k_B \) data are not aligned. To contract \( k^m \), \( k^m_A \) and \( k^m_B \) initially held at \( P[m, y] \) and \( P[x, m] \) need to be held by all nodes. To do this \( P[m, y] \) can broadcast \( k^m_A \) along \( p_x \) to \( P[*, y] \), and \( P[x, m] \) can broadcast \( k^m_B \) along \( p_y \) to \( P[x, *] \). Then, every processor \( P[x, y] \) holds \( k^m_A \) and \( k^m_B \), and a local contraction \( k^m \) can be performed. This is done for each \( m \), so the entire contraction index \( k \) is contracted locally on each node.

This is the scalable universal matrix multiplication algorithm (SUMMA) for matrix multiplication. The algorithm is given in Lst. 3.2 and is depicted in Fig. 3.3. For a contraction involving multiple contraction indices that are DDO, broadcasts are performed recursively along the dimensions of the torus where the contraction indices are distributed; hence, the name recursive broadcast.
3.4 Contraction Algorithm for Symmetric Tensors: CAST

3.4.1 Notation

In this section, we represent tensor blocks as \( \langle i, j \rangle \) and the processor address where they are stored as \( P[I, J] \). Tensor elements are represented as \([i, j]\). To simplify our derivation, we assume the index representing first dimension of the tensor is always mapped to the first dimension of the torus. For example, element \( A[i, j] \) is part of the tensor block \( A\langle i, j \rangle \), which is held at processor \( P[I, J] \). The symbol \( * \) is used to represent all possible values. For example, \( A[* , j] \) is the entire column \( j \) of \( A \).

Figure 3.4: Block-Cyclic Distribution

Figure 3.5: Communication Pattern

3.4.2 Introduction to Symmetric Tensors

A tensor is symmetric within a subset of indices if it is invariant when any two indices of the subset are interchanged. For example, a tensor \( v[a, b, i, j] \) is symmetric on \( a, b \) if \( v[a, b, i, j] = v[b, a, i, j] \). A symmetric tensor can be stored in a compact form: only \( \frac{1}{d!} \) of the elements are stored in memory, where \( d \) is the dimensionality of
the symmetric group. In the example, only $v[a > b, i, j]$ will be stored where $a > b$ implies $a, b | a \geq b$. A tensor can possess multiple symmetric groups of indices. For instance, if $v[a, b, i, j]$ is symmetric on index groups $[a, b]$ and $[i, j]$, then $v[a, b, i, j] = v[b, a, i, j] = v[b, a, j, i] = v[a, b, j, i]$. In this case, there are two symmetry groups of size 2, so the memory reduction is $\frac{1}{2!} \times \frac{1}{2!}$.

Storing symmetric tensors in compact form leads to two main difficulties in contracting symmetric tensors:

- Load balancing data distribution and computation.
- Efficient communication of data: Conventional dense tensor contraction algorithms will not work because symmetric tensors are stored in compact forms.

Load balancing of data distribution and computation can be achieved by using cyclic or block-cyclic distribution. We use block-cyclic distribution for our algorithm for two reasons: 1) it is compatible with existing quantum chemistry suits, and 2) it allows for efficient packing and unpacking of the data required for local computation. In block-cyclic distribution, a tensor is divided into smaller blocks that are distributed along each dimension of a processor grid in a round robin fashion. Consider a two-dimensional (2D) tensor, block cyclically distributed onto an $m \times m$ processor grid. A block represented as $\langle i, j \rangle$ is stored at processor $P[i \% m, j \% m]$, where indices $i$ and $j$ represent row and column indices respectively. Fig. 3.4 shows an example of a block-cyclic distribution of a symmetric matrix $A$ (2D tensor), divided into $8 \times 8$ blocks across a $2 \times 2$ processor grid.

The four processors in the $2 \times 2$ grid are marked in four different colored dots (red, yellow, blue, and green). The block $A(0, 1)$ can be retrieved by transposing the block
$A(1,0)$ stored at the blue processor. Each processor holds several blocks in the grid. Consider block column $k = 1$ of $A$ given by $A(\ast, 1)$. These blocks are distributed among the yellow and green processors. However, the blocks in the $k^{th}$ column of the upper triangular part of $A$ are stored at the blue processor instead of yellow due to symmetry.

Block-cyclic layout of symmetric tensors complicates the communication pattern for tensor contractions. Because only unique blocks of tensors are stored, blocks required to initiate the communication using recursive broadcast are not present at the processor responsible for starting the communication. In this section, we present a novel algorithm that expands on the recursive broadcast scheme to support symmetric tensors.

### 3.4.3 Symmetric Matrix Multiplication

Fig. 3.5 shows an $8 \times 8$ torus. Let tensor $A$ be divided into a $n_A \times n_A$ block grid, which is block cyclically distributed onto the $8 \times 8$ torus. Now, consider all the blocks represented by $\langle \ast, k \rangle$. This collectively gives all the blocks along the $k^{th}$ column of the block grid of $A$. Because $A$ is symmetric, $\langle y, k \rangle = \langle k, y \rangle$, and only one of these blocks is stored. We store only blocks with address $\langle y > k \rangle$. As described before, blocks with address $\langle y, k \rangle$ are stored at $P[Y, K]$ if $y \geq k$. Otherwise, $\langle k, y \rangle$ are stored at $P[K, Y]$, where $Y$ and $K$ are $y \% 8$ and $k \% 8$, respectively. Hence, all the blocks given by $\langle \ast, k \rangle$ are stored at $P[\ast, K]$ and $P[K, \ast]$. This is shown as the red column and row of processors (respectively) in Fig. 3.5.

In conventional dense matrix multiplication algorithms such as SUMMA (recursive broadcast) [132], a processor $P[Y, K]$ broadcasts the blocks it owns to processors
$P[Y, \ast]$, where column $k$ of $A$ is multiplied with row $k$ of $B$. However, in symmetric matrix multiplication, the blocks required are held at $P[Y, K]$ and $P[K, Y]$ (as previously described). Before initiating the broadcast, $P[Y, K]$ can first receive the data for $A$ from $P[K, Y]$ via $P[Y, Y]$ as shown in pink in Fig. 3.5. It can then broadcast the combined data from $P[Y, K]$ and $P[K, Y]$ to $P[Y, \ast]$. Routing the data via the diagonal provides a unique path for each processor in $P[\ast, K]$ to receive the required data from $P[K, \ast]$ without any contention as shown in Fig. 3.5. A similar process can be repeated for $B$.

The process of receiving all of the data required before doing the broadcast is defined as instigation. The processors receiving the data are instigators, and the ones sending the data are senders for instigation. In Fig. 3.5, the processors in the column in red with blue dashed lines are the instigators for $k = 2$, and the processors in the row in red are the senders of that instigation. Thus, a broadcast scheme can be modified to work with symmetric tensors by adding instigation phase before each broadcast.

The algorithm described here only works because the processor grid is a square. For a processor grid of size $m \times n$, block $\langle y, k \rangle$ is held at $P[y \% m, k \% n]$, and block $\langle k, y \rangle$ is held at $P[k \% m, y \% n]$. $P[y \% m, k \% n]$ can be permuted to get $P[k \% n, y \% m]$. However, the required data is in $P[k \% m, y \% n]$. $P[k \% n, y \% m] = P[k \% m, y \% n]$ only if $m = n$. For our algorithm to work, all of the indices of a symmetric group must be mapped to processor dimensions of same size. For the remainder of this section we assume this condition is met. In Section 3.4.6, we show how this condition can be removed from the processor grid. We also assume that each dimension of a tensor is always mapped to a unique dimension of the processor.
grid. This assumption can be trivially met by adding dimensions of size one to any processor grid when enough processor dimensions are not available. This corresponds to serialization of tensor indices, described in the RRR review.

3.4.4 Symmetric 4D contraction

Consider the following symmetric tensor contraction:

\[ C[i > j > l, d] = A[i > j > k > l] \times B[k, d] \quad (3.5) \]

Assume tensor \( A \) is symmetric on indices \([i, j, k, l]\), tensor \( B \) is not symmetric, and \( k \) is the contracting index. Assume the contraction is performed on a four-diensional (4D) processor grid of size \( d \) along each dimension. Each dimension of \( A \) block-cyclically maps to a unique dimension of the processor grid. Tensor \( B \) is block-cyclically distributed across two dimensions and replicated along the remaining two dimensions of the processor grid. Block \( A\langle i, j, k, l \rangle \) is stored at corresponding processors \( P[I, J, K, L] \mid I = i \% d, J = j \% d, K = k \% d, L = l \% d \).

Because there is a 4D symmetry, only blocks with form \( \langle i > j > k > l \rangle \) in \( A \) are stored. As index \( k \) is contracted, processor \( P[I, J, K, L] \) requires blocks of \( A \) collectively represented by \( A\langle i > j > l, k \rangle \) for broadcasting. These blocks can be grouped as \( A\langle k > i > j > l \rangle, A\langle i > k > j > l \rangle, A\langle i > j > k > l \rangle, \) and \( A\langle i > j > l > k \rangle \).

All of the aforementioned blocks are located at 4 processors, \( P[K, I, J, L], P[I, K, J, L], P[I, J, K, L], \) and \( P[I, J, L, K] \), respectively. Processor \( P[I, J, K, L] \) needs to instigate a data collection from all these processors before it can broadcast. The senders send their respective data via the diagonal processors. For instance, data held by \( P[K, I, J, L] \) is sent via \( P[I, I, J, L] \) then via \( P[I, J, J, L] \) to \( P[I, J, K, L] \). In this contraction, instigation is required for each value of the contraction index (as
Listing 3.3: General CAST

```c
CAST(A, B, C, cidxs):
    /* A, B, C : Tensors */
    /* cidxs : contraction indices */
    c = cidxs.pop(0)
    for c in range(0, len(c)):
        if processor_id is instigator of A:
            Collect data in A_recv
            Broadcast A_recv
        if processor_id is sender of A:
            Send data to instigator
        if processor_id is receiver of A:
            Receive broadcasted data A_recv
        /* Repeat for B */
        if len(cidxs) == 0:
            dgemm(A_recv, B_recv, C)
        else:
            CAST(A_recv, B_recv, C, cidxs)
```

described) before a broadcast can be done for A. Since B is not symmetric, it can be directly broadcast.

3.4.5 Generalization of CAST

In the previous example, the presence of a 4D symmetry group in an input tensor required data from four different processors to be combined together before it could be broadcast. For a $n$-dimensional symmetry group, data from $n$ processors needs to be combined together.

Consider an $n$-dimensional tensor: $A[i_1, i_2, ..., i_k, ..., i_n]$ with $n$-dimensional symmetry $i_1 > i_2 > ... > i_k > ... > i_n$ stored at processor $P[I_1, I_2, ..., I_k, ..., I_n]$, where $i_k$ is the contracting index:
$A[i_1 > i_2 > ... > i_{k-1} > i_{k+1} > ... > i_n,k]$ \quad (3.6)

$P[I_1, I_2, ..., I_{k-1}, *, I_{k+1}, ... I_n]$ \quad (3.7)

$P[I_1, I_2, ..., I_{k-1}, K, I_{k+1}, ... I_n]$ \quad (3.8)

$P[\Pi (I_1, I_2, ..., I_{k-1}, K, I_{k+1}, ... I_n)]$ \quad (3.9)

For some particular value $k = i_k$, data collectively represented by Eq. (3.6) which needs broadcast to Eq. (3.7) The data is collected at Eq. (3.8) from processors Eq. (3.9) where $K = I_k$ and $\Pi$ gives all possible permutations of the indices such that $I_{j-1}$ always comes before $I_j$ where $1 \leq j \leq n$ and $I_{k-1}$ always comes before $I_{k+1}$ in the sequence of indices. The lexicographical ordering is preserved among all indices except $K$, to preserve the relative ordering of symmetric indices.

We can now generalize CAST to an arbitrary number of symmetry groups, each with an arbitrary number of symmetric indices. Consider a general tensor $A$ given by:

$A[s_1, s_2, ..., s_m, ns]$ \quad (3.10)

where $m$ is the number of symmetry groups in $A$, $s_j$ is the $j^{th}$ symmetry group, and $ns$ is the set of indices without any symmetry. Since, a symmetry group is simply a set of indices, $s_j$ can be expanded as:

$s_j = [i_{1}^{j} > i_{2}^{j} > ... > i_{d_j}^{j}]$ \quad (3.11)

where $i_{k}^{j}$ is the $k^{th}$ index in $s_j$ and $d_j$ is the total number of indices in $s_j$. Using the notation defined at the beginning of this section, the distribution of blocks of $A$ in
Eq. (3.12) is given by Eq. (3.13) where Eq. (3.14).

\[
A(s_1, ..., s_{j-1}, i_1^j > i_2^j > ... > i_{d_j}^j, s_{j+1}, ..., s_m, ns) \tag{3.12}
\]

\[
P[S_1, ..., S_{j-1}, I_1^j, I_2^j, ..., I_{d_j}^j, S_{j+1}, ..., S_m, NS] \tag{3.13}
\]

\[
S_j = [I_1^j, I_2^j, ..., I_{d_j}^j] \tag{3.14}
\]

For contraction index \(k^j = i_k^j\) in \(A\), the data collectively represented by

\[
A[s_1, s_2, ..., s_{j-1}, i_1^j > i_2^j > ... > i_{k-1}^j > i_{k+1}^j > ... > i_{d_j}^j, s_{j+1}, ..., s_m, ns, k^j] \tag{3.15}
\]

needs to be broadcast among processors represented by Eq. (3.16). Using Eq. (3.9), the data is collected from Eq. (3.17) at processor Eq. (3.18)

\[
P[S_1, ..., S_{j-1}, I_1^j, ..., I_{k-1}^j, *, I_{k+1}^j, ..., I_{d_j}^j, S_{j+1}, ..., S_m, NS] \tag{3.16}
\]

\[
P[S_1^a, S_2^a, ..., S_{j-1}^a, \Pi(I_1^j, ..., I_{k-1}^j, K^j, I_{k+1}^j, ..., I_n^j), S_{j+1}, ..., S_m, NS] \tag{3.17}
\]

\[
P[S_1, S_2, ..., S_{j-1}, I_1^j, I_2^j, ..., I_{k-1}^j, K^j, I_{k+1}^j, ..., I_{d_j}^j, S_{j+1}, ..., S_{m_A}, NS] \tag{3.18}
\]

For a given contraction index \(k^j = i_k^j\), processors given by Eq. (3.16) are receivers, Eq. (3.17) are senders, and Eq. (3.18) are the instigator. Let \(A_{recv}\) be the data represented by Eq. (3.15). The pseudocode for a general symmetric contraction can be represented as Lst 3.3. The structure of this algorithm is similar to that of recursive broadcast. In a recursive broadcast scheme, for a given iteration of the contracting iterator, processors holding the data can initiate a broadcast. For a symmetric tensor, none of the processors initially hold all the data needed for the broadcast. Instigation is done to collect data from senders by instigators. Once data are collected, the broadcast can proceed.

The ability to contract one iteration at a time makes it possible to contract only those iterations that preserve the order of contracting indices. For example, if there
are \( x \) number of contraction indices belonging to the same symmetry group \( j \) represented by \( c_1^j, c_2^j, \ldots, c_x^j \), the contraction needs to be performed only for \( c_1 > c_2 > c_3 \ldots > c_x \). The result can then be multiplied by \( x! \) to obtain the correct result.

### 3.4.6 Using a Virtual Grid for Non-Uniform Physical Grid

CAST works on perfectly square tori, i.e., all the dimension of the physical grid along which symmetric indices are mapped must be equal in size. The sizes do not have to be equal across different symmetry groups, only within the same symmetry group. To overcome this restriction, we introduce a virtual grid. In this scheme, CAST will run on a perfectly square virtual grid that can be mapped to any rectangular physical grid. For instance, a \( 6 \times 6 \) virtual grid may be mapped to a \( 6 \times 3 \) physical grid where each physical node will be performing the work of two virtual nodes. In practice, an instigator processor simply finds out whom its senders are using modular arithmetic and collect data from those senders.

### 3.4.7 Cost of CAST

The cost of CAST is the sum of the cost of instigations and the recursive broadcasts. In the case of a symmetric tensor, if the contraction indices are stored in full, a non-symmetric recursive broadcast scheme can be applied. If the contraction indices are not stored in full, instigation is required at each step of the recursive broadcast. At a single recursion level, representing a particular contraction index, the communication cost for input tensor \( A \) is given by Eq. (3.19).

\[
\text{cost} = (2k + 2p - 3) \left( \alpha + \frac{N_A \cdot p}{P \cdot k} \beta \right) + 2k \cdot (n - 1) \cdot \alpha 
\]  

(3.19)
where \( k \) is the number of blocks along the contraction dimension that is block cyclically distributed among \( p \) nodes, \( \alpha \) is the latency, \( \beta \) is the bandwidth, \( M_A \) is the total size of tensor \( A \) with the contraction index stored in full, and \( P \) is total number of nodes. For example, if there is symmetry between indices, \([i, j, k]\), then we would only store blocks \( \langle i > j > k \rangle \). If \( k \) is a contraction index, then storing it in full means storing blocks \( \langle i > j, k \rangle \). The communication cost for \( B \) can be found similarly. The total communication cost at each recursion level can be added to obtain the total communication cost of performing CAST. The communication cost of CAST achieves the lower bound on communication volume. Derivation of the cost of CAST is provided in [107].

3.5 Redistribution

CAST is an extension to the recursive broadcast algorithm for dense tensors. As described in Section 5.4, it requires contracting indices in input tensors to be distributed distributed orthogonal (DDO). When a sequence of symmetric tensor contractions is computed, e.g., the sequence of contractions in the CCSD equations, the result of a contraction may be an input for the next contraction. In such cases, the input tensors may not always be suitably distributed for CAST across contractions. To get the input tensors in to a suitable distribution, we require a redistribution. In this section we develop an algorithm for efficient redistribution of tensors.

3.5.1 Tensor Distribution

The block-cyclic distribution of a tensor on a processor grid is defined by a block grid, which describes the number of blocks along each dimension of a tensor, and an index-dimension map that maps tensor dimensions to grid dimensions. Both the block
grid and the index-dimension map are vectors of length equal to the dimensionality of the tensor. The redistribution algorithm redistributes a tensor according to its old and new index-dimension map. The details are driven by replication along physical dimensions.

### 3.5.2 Identifying Senders and Receivers

The first step of redistribution is to identify the sender and receiver processors. In our framework, each block of a tensor has a unique vector address. For example, a block in a 4D tensor $A$ can be identified by an address of the form $<i', j', k', l'>$, where $i'$, $j'$, $k'$, and $l'$ represent the position along each index. Given an index-dimension map, dimensionality and size of the processor grid, and the tensor block size, we can determine which blocks a particular processor in the grid holds. Each processor uses the new index-dimension map to find out which blocks it needs to receive, and the old index-dimension map to find where to receive these blocks from. Similarly, each processor can also find where it needs to send the current blocks it holds.

### 3.5.3 Partitioning the Communication Link Usage

Redistribution can be parallelized if the tensor to be redistributed is replicated in the initial distribution. If a tensor is replicated $p$ times, there are $p$ copies of the tensor in the processor grid. Each processor along the replicated dimension is in a different hyperplane, and each hyperplane contains a copy of the entire tensor. If there are $p$ nodes along each dimension of the grid, there will be $p$ such hyperplanes.

Fig. 3.6 shows a 3D processor grid with $p = 4$. One of the grid dimensions has replication along it while the other two dimensions have some tensor index distributed
Figure 3.6: Virtual splitting of a grid into 2D planes along the replicated dimension.

along them. The grid can be virtually split into 2D planes along the replicated dimension. Because all of the dimensions that have some index distributed along them form the hyperplane, each plane contains a copy of the full tensor. Therefore, each hyperplane can handle communication independent of other planes. This greatly simplifies the communication pattern for redistribution and also assures that each processor sends and receives from the nearest processor holding the required tensor blocks.

If there are \( d_r \) replicated dimensions in the grid, the hyperplane splits will occur along all \( d_r \) dimensions. This will yield hyperplanes of \( d - d_r \) dimensions formed with those that have indices mapped to them. Each hyperplane will have a copy of the entire tensor, and communication can be done in each hyperplane independent of others. Similarly, if there is no replication of data in the grid, the entire \( d \)-dimensional grid contains only one complete copy of the tensor. Thus, it can be viewed as a \( d \)-dimensional hyperplane, and there will be no parallel communication.
3.5.4 Redistribution within a Hyperplane

Blocks within a hyperplane will be redistributed either via point-to-point communication when there is no new replicated dimension in the new index-dimension map, or via broadcast when there are replicated dimensions in the new index-dimension map.

Because hyperplanes only consist of dimensions of the grid where tensor indices are mapped based on the old index-dimension map, there is no replicated dimension within a hyperplane. If one or more dimensions of the hyperplane are replicated in the new index-dimension map, the tensor blocks can be redistributed using one-to-all broadcast from the processor that holds these blocks to the set of processors along the replicated dimensions in the new index-dimension map that should receive these blocks.

For example, consider a 3D grid where a processor can be identified as \( P[I, J, K] \). If the first dimension in a 3D grid has replication in the new distribution but not in the initial one, it becomes a broadcast dimension. A broadcast group along this dimension can be represented as \( P[I, *, K] \). If there are \( p \) processors in each dimension of the grid, there will be \( p^2 \) disjoint broadcast groups because \( I \) and \( K \) can take \( p \) possible values and form \( p^2 \) possible combinations for \( P[I, *, K] \).

In general, for a \( d \)-dimensional hyperplane, with \( p \) nodes on each dimension and \( b \) broadcast dimensions, each broadcast group will have \( n^b \) nodes, and there will be \( n^{d-b} \) such broadcast groups. Fig. 3.7 shows the broadcast groups of processors for \( p = 4 \). The groups look like lines of processors (1D) along the first dimension, which is the dimension that will have replication in the new index-dimension map.
In the absence of broadcast communication, point-to-point communication will be done within each hyperplane. The pseudocode for the redistribution algorithm is given in Lst. 3.4.

3.5.5 Generalizing Redistribution

The algorithm presented in this section can redistribute tensors when the index-dimension map changes, but it cannot redistribute if the processor grid also changes. Tensor contractions are usually implemented on virtual grids that are embedded on the actual physical grid using a mapping function. Depending on the type of contraction, it might be necessary to change the virtual grid. For example, in the following sequence of contractions, the first contraction can be computed on a 4D grid, while the second contraction should be computed on a 2D grid to avoid redundant computation when using recursive broadcast algorithm. This requires the redistribution to
Listing 3.4: Redistribution Scheme

grid_dims: grid dimensions
old_idmap: current index-dimension map
new_idmap: new index-dimension map
bcast_dims: list of broadcast dimensions
parallel_dims: list of independent dimensions
hplane_list: list of hyperplanes along replicated dims

for (d in range(0, grid_dims))
  if (d has replication in new_idmap
      AND d has no replication in old_idmap)
    bcast_dims.add(d);

for (i in range(0, grid_dims))
  if (i has replication in old_idmap)
    parallel_dims.add(d);

hplane_list = get_hyperplane_list(parallel_dims);

parallel for each hyper plane in hyper_plane_list
  if (bcast_dims.count > 0)
    redistribute_broadcast(bcast_dims);
  else
    redistribute_point_to_point();

update_idmap(new_idmap);

work when the grid changes (refer to [107] for details).

\[
C[i,j,m,n] = A[i,j,k,l] \times B[k,l,m,n] \quad (3.20)
\]
\[
D[i,n] = A[i,k,l,o] \times B[k,l,o,n]
\]

The redistribution can be done by extending the algorithm presented herein. As before, we split the old processor grid based on replications in the old index-dimension map. However, to find senders and receivers for communication within a hyperplane, the new index-dimension map and the new processor grid are used. As before, we use broadcast if there is replication within the hyperplane. Otherwise we do point-to-point communication.
Figure 3.8: Redistribution of $A(i')$ from $i \rightarrow x$ to $i \rightarrow y$: the left figure shows the redistribution scheme described in this section while the right figure the naïve redistribution scheme.

3.5.6 Key Insight

In this subsection, we summarize the highlights of our redistribution scheme and contrast it with a naïve implementation of redistribution. In a naïve scheme, for each data block, a unique processor holding that data block is identified, and that processor will be responsible for sending the data to all the processors that hold it in the new distribution. If there are multiple processors holding that data block, the processor with the smallest lexicographical address is chosen as the sender. Hence, a unique sender is identified along with all of its receivers for each data block. Then a many-to-many communication is performed.

Our redistribution scheme has two major advantages over the naïve scheme:

- Partitioning the communication link usage: If there is replication in the old index-dimension map, we partition the grid into hyperplanes along the replicated dimension. Unlike the naïve scheme, because each hyperplane holds an entire copy of the tensor, it is responsible for communication only within itself.
For each data block, there are as many senders as the number of replicated copies of that data block. Each sender is only responsible for communicating with receivers within its hyperplane.

- Identifying collectives: If there are replicated dimensions in the new index-dimension map, but not in the old index-dimension map then these replication dimensions are recognized as broadcast dimensions. The naïve scheme does not identify such collective operations.

3.5.7 Clarifying example

Consider a $3 \times 3$ processor grid with dimensions $x$ and $y$. A 1D vector $A[i]$ is mapped on to this grid such that $i \rightarrow x$ and $A$ is replicated along $y$. Let the new index-dimension map be $i \rightarrow y$, and $A$ is replicated along $x$. $A[i']$ represents a block of $A$ and let $i' \in \{0, 1, 2\}$. Fig. 3.8 shows how redistribution is done using our scheme and the naïve scheme. The blue, green, and pink blocks in the figure correspond to $A(0), A(1)$, and $A(2)$, respectively, and are replicated along the $y$ dimension as described by the index map. In our scheme, the grid is divided into three 1D hyperplanes, and each hyperplane contains the entire tensor. The new index-dimension map has replication along $x$ dimension. This is identified as a broadcast dimension, and data are broadcast independently within each hyperplane shown by horizontal blue, green and pink bidirectional arrows. Notice that no communication is done across hyperplanes. In the naïve scheme, all the senders are from the same hyperplane, and a many-to-many is performed. Communication across the hyperplanes is also required.
Figure 3.9: The left figure shows the communication and computation time for seven contractions on 16,384 BG/Q nodes (262,144 cores), while the right figure shows times on 1,024 Cray XE6 nodes (24,576 cores).

3.6 Experiments

In this section, we compare the computation time, communication time, and scalability of CAST to that of the CTF [118]. We also show redistribution performance for point-to-point redistributions with and without parallel hyperplanes.

3.6.1 Communication Time

We compare our implementation with CTF [118] on seven representative examples of symmetric tensor contractions from CCSD equations and collect results on two supercomputers: IBM Blue Gene/Q (BG/Q) and Cray XE6. The BG/Q consists of 16 cores (1.6 GHz) and 16 GB of memory per node, with a 5D torus interconnect. The Cray XE6 consists of dual socket 12-core AMD Magny-Cours (2.1 GHz) and 32 GB of memory per node with a 3D torus Cray Gemini interconnect. However, the Cray XE6 does not support topology-aware mapping. Our code is implementation
in C/C++ and uses Message Passing Interface (MPI) for communication. Computationally expensive routines are threaded using OpenMP. We use vendor-optimized basic linear algebra subroutines (BLAS) implementation for computation (IBM ESSL and Cray LibSci).

Fig. 3.9 shows the communication and computation time of CAST and CTF for problem size N=384 on BG/Q and N=256 on Cray XE6 (N is the size of each dimension of the tensor) for all contractions-except contraction 6 where N=32768 on the BG/Q and Cray XE6 because its computational cost is much smaller. On BG/Q,
Table 3.1: List of Contractions

1 \[ C[i, j, a, b] = A[i, k; a > l] \times B[l, j, k > b] \]

2 \[ C[i > j, a > b] = A[i > k, j > l] \times B[a > b, k > l] \]

3 \[ C[i, a, j, b] = A[i > k, j > l] \times B[l, a, k, b] \]

4 \[ C[i, a, j, b] = A[i > k, j > l] \times B[l, a, k > b] \]

5 \[ C[i, a, j, b] = A[i > k, j > l] \times B[l > a, k > b] \]

6 \[ C[i, a] = A[i > k] \times B[k > a] \]

7 \[ C[i, j, a, b] = A[i, j, b > k] \times B[a > k] \]

we tested on 16,384 nodes (262,144 cores) with one MPI rank per node. On the Cray XE6, we test on 1,024 nodes (24,576 cores) with four MPI ranks per node. We use DGEMM to denote the time spent on BLAS computation, and COMM for the remaining time, which is primarily composed of communication time.

3.6.2 Scalability

The seven contractions described in Table 3.1 can be categorized as 4D-4D (Contractions 1,2,3,4,5), 2D-2D (Contraction 6), and 4D-2D (Contraction 7). Contraction 2 is the only contraction with symmetry in the external indices in the 4D-4D category. Contractions 1, 3, 4, and 5 are similar contractions with varied degrees of symmetry. We show a scalability comparison of CAST to CTF using Contractions 2, 3, 6, and 7.

Fig. 3.10 shows the scalability of CAST and CTF on BG/Q. The slope of the scalability curve for CAST does not alter drastically for any contraction. CAST does
Figure 3.11: Scalability of Contractions 2, 3, 6, and 7 up to 1024 Cray XE6 nodes for $N=192, 256, \text{ and } 32768$.

not do any redistribution or transposes during its symmetric contractions. Transposes within a contraction require global barriers. By avoiding global barriers and the communication required for transposes, CAST can scale up to 16,384 nodes (262,144 cores). For CTF the scalability trends look good for Contractions 6 and 7. However, the slopes change directions for Contractions 2 and 3 on 4,096 and 16,384 nodes because of large packing and unpacking overhead, and expensive all-reduce algorithm that CTF uses at these torus sizes. Fig. 3.11 shows the scalability of CAST and CTF on Cray XE6. Both CAST and CTF have similar slopes with different offsets. CAST outperforms CTF on all but Contraction 2. The block-cyclic distribution that CAST
uses can incur a slight load imbalance when the resulting tensors are symmetric, which results in slightly lower performance for CAST on Contraction 2. We note that the CAST approach could also be implemented with a fully cyclic distribution to improve load balance, but we would then lose compatibility of representation with production parallel quantum chemistry suites, such as NWChem.

Overall, CAST outperforms CTF in most contractions. However both CAST and CTF both have strengths and weaknesses that are apparent in Fig. 3.9, Fig. 3.10, and Fig. 3.11. The most optimal scheme could be something that combines the strengths of both, which could be a future research direction.

3.7 Related Work

Due to the prominent role of tensors in quantum chemistry, efficient execution of tensor contraction expressions have been studied [13]. Efforts to minimize the number of operations in chains of tensor contraction expressions [57], minimize their memory requirement [25], and trade off increased computation cost to reduce space [23] have all been considered. None of these efforts directly attempted to minimize the communication cost.

Cociorva et al. [24] considered minimizing the communication volume using the Cannon’s algorithm in the context of multiple tensor contractions under memory constraints. Gao et al. [48] extended this algorithm by accounting for disk input/output (I/O) costs. These algorithms only employed a 2D processor grid and did not handle permutation symmetry in tensors.
NWChem [130] is a production computational chemistry suite that implements a number of accurate high-order models whose computationally intensive operations are tensor contractions. The tensors are stored as a collection of full “bricks,” distributed over the system nodes using the Global Arrays (GA) toolkit. A simple dynamic load balancing scheme is employed. The loops representing a tensor contraction expression are tiled and parallelized, akin to an OpenMP parallel loop. Each processor dynamically determines the set of brick-brick contractions to perform and executes a remote get (using GA calls) for each input brick operand and local tensor contraction. A remote associative update (using GA) is then performed on the result brick. Thus the current implementation only focuses on dynamic load balancing of the work in performing distributed tensor contractions without any emphasis on reducing communication costs. While this was considered good enough in the past, recent work has shown that communication costs now impose fundamental limits on scalability of these methods. Kowalski et al. [75] observed that while the computationally dominant ($O(N^7)$) non-iterative triples calculation scales to hundreds of thousands of cores, the less expensive ($O(N^6)$) iterative portion does not scale beyond a few thousand cores. This disparity between the two sets of tensor contractions has resulted in the less expensive iterative calculation requiring greater runtime than the non-iterative calculation, thus becoming the computational bottleneck. Lai et al. [80] partially addresses this problem by exploiting parallelism across multiple contractions but does not optimize communication. Our work in this dissertation is aimed at ultimately solving the scalability problem of NWChem, but requires the incorporation of block-sparsity in the distributed tensor contraction before it can do so.
Solomonik et al. [118] created the first implementation of communication-optimized and load-balanced distributed contraction algorithm for symmetric tensors. Their CTF uses a cyclic distribution of tensors across a multidimensional physical torus, thereby ensuring excellent load balancing of all tensors. Contractions involving symmetric tensors are achieved using a number of passes of a generalization of the 2D SUMMA [132] algorithm, with dynamic redistribution of symmetric tensors in between passes.

Schatz [112] has been developing an “Anatomy of Parallel Computation with Tensors”, a study somewhat similar to the RRR framework in Rajbhandari et al. [107, 105]. Schatz’s work focuses on finding a relationship between the distribution of tensors and a set of collective communications that can be applied for the computation. Conversely, RRR focuses on using reuse dimensions as communication directions and the use of appropriate collective operations to achieve communication along these directions. Both works attempt to deconstruct tensor computation into its fundamental components.

Communication lower bounds and associated algorithms have been studied for various configurations of 2D matrix multiplication operations [65, 9]. In particular, 2D algorithms have been shown to incur asymptotically higher communication costs than algorithmic lower bounds. 3D algorithms, which are communication optimal, incur higher memory overheads. Solomonik and Demmel [120] presented the 2.5D algorithm, which is communication efficient [119] and considers a trade off between memory costs communication minimization. CAST differs from CTF in three significant ways: 1) it avoids the need for multiple dynamic redistributions when contracting
symmetric tensors; 2) it effectively uses parallelism across hyperplanes while performing redistribution; 3) it uses a block-cyclic distribution of tensors instead of a fully cyclic distribution, allowing this implementation to be compatible with the internal blocked representation of tensors used in production parallel quantum chemistry suites, such as NWChem.

3.8 Conclusion

We presented a novel and communication-efficient algorithm for contracting tensors with symmetry. We also presented a communication-efficient redistribution algorithm. These algorithms were used to demonstrate CAST’s scalability on BlueGene/Q and Cray XE6 systems. As future work, we are considering the use of CAST in scaling production computational software suites, such as NWChem, on massively parallel supercomputers.

Acknowledgment

This work was supported in part by the U.S. Department of Energy’s (DOE) Office of Science, Office of Advanced Scientific Computing Research, under awards 63823 and DE-SC0008844.
Chapter 4: Optimizing 4-index Transform

4.1 Introduction

The four-index integral transform is a computationally demanding calculation implemented in numerous computational chemistry suites, including NWChem [131], ACES [126], GAMESS [114], MOLPRO [138], MPQC [68], and PSI [129]. It is used to transform a four-dimensional tensor of coefficients ($A$) from an atomic basis to a molecular basis, using repeated applications of a two-dimensional transformation matrix $B$. The computation can be very simply expressed as follows:

$$C[\alpha, \beta, \gamma, \delta] = \sum_{i,j,k,l} A[i, j, k, l].B[\alpha, i].B[\beta, j].B[\gamma, k].B[\delta, l]$$  \hspace{1cm} (4.1)

However, efficient implementation of this transformation is non-trivial and has been the subject of numerous efforts [86, 46, 140, 142, 141, 44]. The direct conversion of the above expression into code would result in an implementation with a computational complexity of $O(n^8)$, if all eight indices ($\alpha, \beta, \gamma, \delta, i, j, k, l$) range from 1 to $n$. By utilizing algebraic properties of associativity, commutativity, and distributivity, the transformation can be implemented with a computational complexity of $O(n^5)$ (as discussed in greater detail in the next section) as a sequence of four steps, each involving the tensor product of a 4D tensor with the 2D transformation matrix $B$. 

82
There are a huge number of possible ways of partitioning the data and computation across a parallel system, along with fusion/tiling choices to reduce data movement through the memory hierarchy. Further complicating matters is the fact that the tensors have varying degrees of permutation symmetry (elaborated in the next section). The storage savings from only representing distinct elements must be exploited, which further complicates reasoning about the data movement cost of alternative distributed implementations and the numerous fusion/tiling choices. Over the years, various versions of the four-index transform have been implemented. In the next section we discuss some of the most commonly used variants within NWChem, representing different space-time trade-offs. A key challenge is that the size of intermediate tensors in the four-step index transformation are larger than the final result. This means that either judicious fusion across the steps is required to optimize memory (which complicates the implementation) or that memory use is sub-optimal, limiting the size of the systems that can be modeled.

In this chapter, we address the development of a parallel high-performance implementation of the four-index transform. We pursue a novel approach using data movement lower bounds to guide the optimization process. The standard approach to developing optimized implementations is either to develop a parametric performance model and find optimal value of parameters, or use empirical auto-tuning across the parameter space. For the four-index transform, the combination of the number of fusion choices and tile sizes along the loops is prohibitively high. Each tensor contraction in the sequence of four contractions has five nested loops. For each adjacent pair of contractions, four of the five nested loops are common and any combination of these can be selected for fusion. It is also possible to fuse a subset of three common
loops across a contiguous set of three contractions, or one of two common loops across all four contractions. Further, we must consider tiling choices for loops. Finally, the optimal configuration may not be the same for different problem sizes, i.e., a one-time expensive auto tuning run that executes thousands of configurations is not feasible - the conclusions may not hold for a different problem size chosen by the user. Thus, neither model-based optimization, nor any successful auto-tuning approach has been previously reported for the four-index integral transform.

We avoid the combinatorial explosion in the number of code configurations by using a novel and completely different approach to guide the optimization: using lower bounds on data movement complexity. Lower bounds on data movement (explained in the next section) are schedule-independent assertions on the minimal amount of data movement required for any valid implementation of the specified computation (modeled as a computational directed acyclic graph). By reasoning about the relationship between the lower bounds on data movement for a single tensor contraction and that of a sequence of contractions, we are able to establish necessary conditions for fusion across a set of contractions to be useful. This allows us to prune away fusion choices that are provably not very beneficial, irrespective of the tile sizes chosen for the fused implementation.

We use lower bounds analysis to model the following two specific optimization scenarios:

1. **Minimizing data movement:** Given a two-level memory view with limited fast “local” memory and unbounded but slow “global” memory, which sets of tensor contractions should be fused to minimize data movement between the limited fast memory and the plentiful slow memory? Such modeling is useful both for
minimizing “vertical” data movement through the levels of the cache/memory hierarchy on a single node, as well as “horizontal” movement between the local memories of nodes in a cluster.

2. **Maximizing size of systems handled:** Given a distributed-memory system with a given collective physical memory, how should the sequence of tensor contractions be fused so as to maximize the size of the systems that can be processed without requiring any out-of-core computation and file I/O (because file I/O on supercomputers is several orders of magnitude slower than inter-processor communication)?

Using the above characterization, we proceed to develop two parallel implementations for the four-index integral transform: 1) A first parallel implementation that maximizes the size of atomic integral tensors that can be transformed without resorting to space-time trade-offs and the associated overheads from redundant recomputations; and 2) A second version that performs further fusion on the node code from the first scheme, thereby reducing inter-processor communication. Both versions are evaluated on two parallel clusters with widely differing architectural characteristics and reveal interesting trade-offs - the former is consistently better on one system, while the latter is generally better on the other cluster.

The chapter makes the following contributions:

- For sequences of loops representing a producer-consumer relationship, it develops the first approach to use data movement lower bounds to choose between loop fusion choices to minimize data movement (**Sec. 3**) and its application to find an optimal fusion choice for the four-index integral transform (**Sec. 4**).
• It describes the first use of data movement lower bounds to reason about minimal space requirements for I/O-free computations for the four-index integral transform (Sec. 5).

• It develops new parallel implementations (Sec. 6) of the four-index integral transform that improve upon the current state-of-the-art implementations in production computational chemistry codes in two ways: 1) For a given amount of total collective physical memory in a cluster, it can run the provably largest four-index transform that does not use file I/O or redundant re-computation of intermediate tensors; 2) For a given amount of per-process local memory, it minimizes the data movement between non-local and local memory.

• It presents an experimental evaluation of the new parallel implementations, demonstrating significant performance improvement over the best current options in the production NWChem computational chemistry suite (Sec. 7).

4.2 Background

In this section, we provide necessary background information before describing the new contributions. We first elaborate on the four-index integral transform and its implementations in NWChem, followed by a brief introduction to data movement lower bounds.

4.2.1 NWChem and four-index transform

NWChem is widely regarded as one of the computational chemistry software suites implementing the most scalable and state-of-the-art implementations of quantum chemistry models. Since it is open source software with a wide user and contributor
base, we use its implementations as our reference point and develop our optimized implementation of the four-index transform within the NWChem framework.

In equation 4.1, if all indices \((\alpha, \beta, \gamma, \delta, i, j, k, l)\) range over the integer interval \([1, n]\), a naive implementation of the transform would take \(5n^8\) operations. In practice, it is implemented as a sequence of four tensor contractions (shown in Equation 4.2). Each tensor contraction (higher dimensional analogs of matrix-matrix product) has a computational cost of \(2n^5\) operations. The output tensor of a contraction serves as an input tensor for subsequent computation.

\[
\begin{align*}
O_1[\alpha, j, k, l] &= \sum_i A[i, j, k, l].B[\alpha, i] \\
O_2[\alpha, \beta, k, l] &= \sum_j O_1[\alpha, j, k, l].B[\beta, j] \\
O_3[\alpha, \beta, \gamma, l] &= \sum_k O_2[\alpha, \beta, k, l].B[\gamma, k] \\
C[\alpha, \beta, \gamma, \delta] &= \sum_l O_3[\alpha, \beta, \gamma, l].B[\delta, l]
\end{align*}
\]

One can see that for each adjacent pair of tensor contractions, four of the five loops iterate over common tensor indices, while one loop differs. Thus, many possible fusion choices exist, where loops over the same tensor indices are fused across two or more consecutive contractions.

**Symmetry in tensors.** Tensors in the four-index transform exhibit permutation symmetry. A tensor is symmetric with respect to a subset of its indices if permuting the indices within the subset does not change the value of the tensor. As an example, consider a tensor \(V[a, b, i, j]\). We say that indices \(a\) and \(b\) are symmetric if \(V[a, b, i, j] = \)
A symmetric tensor can have multiple symmetry groups. For example, we say \( a, b \) and \( i, j \) are two symmetry groups of \( V \) if 
\[
V[a, b, i, j] = V[b, a, i, j] = V[b, a, j, i] = V[a, b, j, i].
\]
This tensor can be stored in a compact form \( V[a < b, i < j] \) due to its symmetry, and can be represented in compact form as \( V[ab, ij] \). The symmetry properties imply that only a fraction \( \frac{1}{d!} \) of the elements of the full tensor need to be stored in the memory, where \( d \) is the number of dimensions in a symmetry group. In case of \( V \), there are two symmetry groups of size 2, hence only a fraction \( \frac{1}{2} \times \frac{1}{2} \) of the elements need be explicitly stored.

Permutation symmetry in the tensors in equation 4.2 can be represented as \( A[ij, kl] \), \( O1[\alpha, j, kl] \), \( O2[\alpha, \beta, kl] \), \( O3[\alpha, \gamma, l] \) and \( C[\alpha, \beta, \gamma, \delta] \). In addition to permutation symmetry, tensors in molecular basis exhibit spatial symmetry. Spatial symmetry is a structured form of sparsity that causes certain blocks to be zero if the index ranges for that block (corresponding to specific molecular orbitals) satisfy a condition. Whereas permutation symmetry reduces the size of all tensors—input, intermediate, and output—in the four-index, spatial symmetry results in additional size reduction of the final output tensor \( C \). The sizes of individual tensors for the four-index transform are shown in Table 4.1.

**Tensor data structures in NWChem.** In NWChem, tensors are blocked along each dimension, resulting in a set of data-tiles representing the tensor. These data-tiles are linearized and distributed using Global Arrays [97, 77], a global address space framework that provides a logical shared view of arrays in a distributed-memory environment.

\(^1\) Tensors in quantum chemistry generally possess anti-symmetry rather than symmetry, i.e., 
\[
V[a, b, i, j] = -V[b, a, i, j],
\]
and our codes implement anti-symmetry. However, for simplicity, we use symmetric tensors in our presentation.
Listing 4.1: Unfused 4-index transform

```c
/*Memory required: n^4, Computation : 4n^5*/
malloc(A,n^4/4); malloc(O1,n^4/2)
for a, i, j, k>l
  O1[a,j,k>l] += A[i>j,k>l] * B[a,i] //n^5
delete(A); malloc(O2,n^4/4)
for a>b, j, k>l
  O2[a>b,k>l] += O1[a,j,k>l] * B[b,j] //n^5
delete(01); malloc(O3,n^4/2)
for a>b, c, k, l
  O3[a>b,c,l] += O2[a>b,k>l] * B[c,k] //n^5
delete(01); malloc(O3,n^4/32)
for a>b, c>d, l
  C[a>b,c>d] += O3[a>b,c,l] * B[d,j] //n^5
```

Listing 4.2: First two and last two fused

```c
/*Memory required: n^4/2, Computation : 4n^5 */
malloc(A,n^4/4); malloc(O1,n^2)
malloc(O2,n^4/4)
for (k>l)
  for (a,i,j)
    O1[a,j] += A[i>j,k>l] * B[a,i] //n^5
for (a>b, j)
  O2[a>b,k>l] += O1[a,j,k>l] * B[b,j] //n^5
delete(A); delete(O1)
malloc(O3,n^2); malloc(C,n^4/32)
for (a>b)
  for (c,k,l)
    O3[a>b,c,l] += O2[a>b,k>l] * B[c,k] //n^5
for (c>d,l)
  C[a>b,c>d] += O3[a>b,c,l] * B[d,j] //n^5
```

Listing 4.3: With re-computation

```c
/*Output streamed. Memory required : n^3, Computation ← n^8+ */
malloc(C,n^4/32)
for (a>b, c>d)
  malloc(O1,n^3)
  for (i,j,k>l)
    O1[i,j,k>l] += A[i>j,k>l] * B[a,i] //n^8
  malloc(O2,n^2)
  for (j,k>l)
    O2[k>l] += O1[i,j,k>l] * B[b,j] //n^7
  delete(O1); malloc(O3,n)
  for (k,l)
    O3[l] += O2[k>l] * B[c,k] //n^6
  delete(O2,n^2)
  for (l)
    C[a>b,c>d] += O3[l] * B[d,j] //n^5
```
Table 4.1: Sizes of intermediate tensors. $s$ is the reduction due to spatial symmetry in tensor $C$.

<table>
<thead>
<tr>
<th>Tensors</th>
<th>Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$n^4/4$</td>
</tr>
<tr>
<td>O1</td>
<td>$n^4/2$</td>
</tr>
<tr>
<td>O2</td>
<td>$n^4/4$</td>
</tr>
<tr>
<td>O3</td>
<td>$n^4/2$</td>
</tr>
<tr>
<td>C</td>
<td>$n^4/(4s)$</td>
</tr>
</tbody>
</table>

parallel system. Each parallel process in a parallel program can get, put, or additively update any arbitrary tile in the tensor.

During a contraction, each distributed processor node is responsible for computing a subset of the tensor tiles. To this end, each processor requests the input tiles required for computing the owned output tile using $GA_{Get}$ and the corresponding tile offsets. After computing the output tile, the processor then stores/updates tiles using $GA_{Put}$.

Lst. 4.4 depicts the implementation the first tensor contraction in the four-index transform as performed in NWChem. Here, loop indices represent tile indices in different tensor dimensions. Line 9 checks if a current tile is owned by the processor. If the tile is owned, then lines 12 and 13 request the required input tiles using $GA_{Get}$. Line 16 stores the computed result using $GA_{Put}$.

**Four-index transform variants in NWChem.** The four-index transformation is an integral part of all post Hartree-Fock calculations, where a molecular representation of two-electron integrals is required. The best example is provided by the coupled cluster formalism (CC), where molecular two-electron integrals are needed.

```c
int node_id = GA_Nodeid();
for l, k, j, alpha
  if (node_id owns tile C[alpha,j,k,l])
    malloc (C_buf, Tilesize)
    for i
      A_buf = GA_Get(Tile A[i,j,k,l])
      B_buf = GA_Get(Tile B[alpha,i])
      DGEMM(A_buf, B_buf, C_buf)
    GA_Put(C_buf to tile C[alpha,j,k,l])
```

to calculate cluster amplitudes and CC energies. The NWChem implementations of CC formalisms fully rely on the complete form of transformed molecular 2-electron integrals. To this end, NWChem has several implementations of the four-index transformation:

**unfused:** Fully unfused version, where $(ij|kl)$, $(\alpha j|kl)$, $(\alpha \beta|kl)$, and $(\alpha \beta|\gamma l)$ tensors are created, which results in $N^5$ scaling of the algorithm but also leads to high global memory requirements. This is shown in Lst. 4.1.

**12-34 fused:** Fused algorithms that do not require the storage of memory consuming $(\alpha j|kl)$ and $(\alpha \beta|\gamma l)$ intermediates. This is shown in Lst. 4.2.

**recompute:** This version computes batches of 2-electron integral $((ij|kl)^{(M)}, \ M = 1, M_B)$. This option allows one to reduce the memory footprint at the cost of computations. Generally, this option will be more time consuming than the but has the lowest memory requirement. This is shown in Lst. 4.3.

### 4.2.2 Data movement lower bounds

Consider matrix-matrix multiplication

$$C[i,j] = A[i,k] \times B[k,j]$$  \hspace{1cm} (4.3)
The standard $O(N^3)$ algorithm is shown in two forms in Fig. 4.1: (a) basic untiled form and (b) tiled form. Both versions require the same number ($2N^3$) of arithmetic operations to compute. But what about the number of data movements through a hierarchical memory system, consisting of main memory (also possibly disk, for the out-of-core case), multiple levels of cache, and/or scratch-pad memory, and registers at the highest level?

Let us consider the simple memory hierarchy containing two levels, with all matrices fitting within the latter level, but too large to fit in the level closer to the processor, which can store only $S$ data elements, $S < N^2$. The untiled version requires the entire matrix $B$ to be accessed for each iteration of the outer $i$-loop. Therefore, the number of data movement operations from the latter level of memory to the closer level for the untiled version is at least $N^3$ (ignoring data movement for $A$ and $C$). However, the tiled version needs much less data movement, $O\left(\frac{N^3}{B}\right)$, for $B < \sqrt{\frac{S}{3}}$.

**Figure 4.1:** Matrix-matrix multiplication kernel: untiled (top) and tiled (bottom)
of data movement can depend on: i) which version of the code is used, ii) the size $S$ of fast memory, and iii) the tile size $B$. A question of interest is: What is the minimum possible amount of data movement among all equivalent forms of code, to compute the standard matrix-matrix multiplication algorithm?

In general, this question cannot be answered without examining a combinatorially explosive number of possible valid schedules for a given computation. The foundational work by Hong and Kung [61] developed an approach to provide lower bounds on the data movement required by any valid execution of the set of operations for a given algorithm. The key idea of their approach was the establishment of an equivalence between any valid schedule for a computation and a constrained graph partitioning problem on the computational directed acyclic graph (CDAG) representing the operations and inter-operation dependences. Given any CDAG, a lower bound on the minimal data movement for its execution has a direct relationship to the minimum number of vertex-sets in an $S$-partitioning of the CDAG graph. In this paper, we use Elango et al.’s [41] adaptation of Hong/Kung’s $S$-partitioning model for the restricted execution model where repeated redundant computation of values (in order to save data movement costs) is prohibited. Although such a model is more restrictive than the original model by Hong/Kung [61], Elango et al. showed that it was feasible to develop tighter lower bounds under such a model for CDAGs.

Hong and Kung showed that on a system with a two-level memory hierarchy with fast memory capacity of $S$ elements, the minimum I/O for multiplying two $n \times n$ matrices is $\Omega(n^3/\sqrt{S})$. Irony et al. [66] provided a lower bound with scaling constants for multiplication of an $n_i \times n_j$ matrix and an $n_j \times n_k$ matrix to be
\( \Omega(n_i \times n_j \times n_k/(2\sqrt{2S}) \). A much tighter bound was provided by Dongarra et al. \cite{40}:
\( \Omega(1.73n_i \times n_j \times n_k/\sqrt{S}) \).

Tensor contractions represent a generalization of matrix-matrix multiplication. We use the lower bound complexity results for matrix-matrix multiplication in exploring fusion/tiling choices for the four-index integral transform computation.

**Wildcard Notation:** \( A[a*, b] \) in Listings mean all indices of \( a \). Similarly, \((alpha > beta)*\) means all \( alpha \) and \( beta \) such that \( alpha > beta \).

### 4.3 Utility of Fusion

In this section, we formalize the notion of utility/futility of fusion for reducing I/O between slow and fast memory for a producer-consumer operation pair. We show that fusion is only useful when the intersection of I/O between a producer operator \( op_1 \) and consumer operator \( op_2 \) is of the same order as the tight lower bounds on the I/O of individual operators. This result is presented in Corollary 4.3.4. We use this result in analyzing loop fusion for the four-index transform. However, this result is general and can be applied to any producer-consumer operation pair.

#### 4.3.1 From tight lower bound to futility of fusion

Here, we establish the utility/futility of fusion in reducing I/O between a fast (but limited capacity) memory and a slow memory.

**Definition 4.3.1 (fusion).** Let \( op_1 \) and \( op_2 \) be two computations where the output of \( op_1 \) is consumed by \( op_2 \). A non-fused schedule of \( op_1 \cup op_2 \) is a schedule where all operations constituting \( op_1 \) are fully completed before \( op_2 \) starts. A fused schedule is
a schedule where some interleaving of the constituent elementary operations of op1 and op2 occur.

Let $f_{12}$ be a fused schedule for op1 and op2, and let $Q_{12} = IO_{LB}(op1 \cup op2)$ be a lower bound on I/O for $f_{12}$.

**Definition 4.3.2** ($\epsilon$-Futility/usefulness of Fusion). A fused schedule $f_{12}$ is said to be $\epsilon$-futile if

$$
(\text{IO}_{\text{opt}}(op1) + \text{IO}_{\text{opt}}(op2)) - Q_{12} \leq \epsilon \left( \text{IO}_{\text{opt}}(op1) + \text{IO}_{\text{opt}}(op2) \right)
$$

$f_{12}$ is said to be useful if $\nexists \epsilon \ll 1$ s.t. $f_{12}$ is $\epsilon$-futile.

By choosing a reasonable (not too small) value for $\epsilon$, this definition of futility of fusion requires that the reduction in I/O from fusion be of the same order as $\text{IO}(op1) + \text{IO}(op2)$. If the reduction in I/O is not of the same order as $\text{IO}_{\text{opt}}(op1) + \text{IO}_{\text{opt}}(op2)$, then we cannot expect to see practical performance gains from fusion. In such scenarios, it is not worth performing loop fusion across the producer-consumer operation pair.

Let $I_1$ and $I_2$ be the set of all elements in the input data spaces of op1 and op2, respectively. Let $O_1$ and $O_2$ be the set of all elements in the output data spaces of op1 and op2, respectively. Let $IO_{LB}(op1)$ and $IO_{LB}(op2)$ be some lower bounds on the number of I/O for separate execution of op1 and op2, respectively. The following lemma allows us to express an I/O lower bound for a fusion of op1 with op2 in terms of $IO_{LB}(op1)$ and $IO_{LB}(op2)$.

**Lemma 4.3.3.** A lower bound on the I/O of any fused schedule for computing $op_{12} = op1 \cup op2$ is given by

$$
IO_{LB}(op_{12}) = IO_{LB}(op1) + IO_{LB}(op2) - |I_1 \cap I_2| - 2|O_1 \cap I_2|
$$
Proof. \( I_1 \cup O_1 \) gives the union of points in the input and output data spaces of \( op_1 \). 
\((I_1 \cup O_1) \cap I_2\) are the set of all points in the input and output data space of \( op_1 \) that are also contained in the input data-space of \( op_2 \). This set corresponds to the data both used by \( op_1 \) and \( op_2 \). From a fused schedule \( f_{12} \) of minimal I/O \( IO_{opt}(op_{12}) \), we can extract a valid schedule for \( op_1 \) that completes its operations by writing to memory (if not already done) all elements of \( O_1 \cap I_2 \). Similarly we can extract a valid schedule for \( op_2 \) that starts by reading from memory all elements of \( I_2 \) that where reused in \( f_{12} \). Summing the I/O of the thus constructed unfused schedules gives \( IO_{opt}(op_{12}) + |I_1 \cap I_2| + 2|O_1 \cap I_2| \), which by definition has to be larger than the sum \( IO_{LB}(op_1) + IO_{LB}(op_2) \) of individual lower bounds.

Corollary 4.3.4. Fusion \( f_{12} \) is useful only if

\[
IO_{opt}(op_1) + IO_{opt}(op_2) = \mathcal{O}\left(|(I_1 \cup O_1) \cap I_2|\right)
\]

Proof. Taking \( IO_{opt}(op_1) \) and \( IO_{opt}(op_2) \) as lower bounds, we get from Lemma 4.3.3 that \( IO_{LB}(op_{12}) = IO_{opt}(op_1) + IO_{opt}(op_2) - |I_1 \cap I_2| - 2|O_1 \cap I_2| \) is a lower bound for \( op_{12} \). Replacing \( Q_{12} \) by this expression of \( IO_{LB}(op_{12}) \) in Def. 4.3.2 from Lemma 4.3.3, we get that fusion is \( \epsilon \)-futile if \( |I_1 \cap I_2| + 2|O_1 \cap I_2| \leq \epsilon \left(IO_{opt}(op_1) + IO_{opt}(op_2)\right) \).

Now for \( f_{12} \) to be useful, there should not exists \( \epsilon \ll 1 \). For any \( \epsilon = \Omega(1) \), \((|I_1 \cap I_2| + 2|O_1 \cap I_2|)/\epsilon \geq IO_{opt}(op_1)+IO_{opt}(op_2)\). Hence, \( IO_{opt}(op_1)+IO_{opt}(op_2) = \mathcal{O}\left(|(I_1 \cup O_1) \cap I_2|\right) \).

4.4 Analysis of fusion: four-index transform

In this section, we show that it is useful to fuse any pair of consecutive tensor contractions in the four-index transform only if \( S = \Omega(n^2) \), where \( n \) is the extent
of each dimension of the tensors. We show that under this condition, the I/O lower bound on a fused pair of tensor contractions given by Lemma 4.3.3 is a tight bound. This result is summarized in the following frame.²

| Given: Two consecutive tensor-contractions $op1$ and $op2$ |
| Problem: Can fusion of $op1$ and $op2$ help in reducing $IO$? |
| Result1: Fusion is useful only if $S = \Omega(n^2)$ |
| Result2: If $S = \Omega(n^2)$ then $IO_{opt}(op1 \cup op2) = |I1| + |O2|$ |

Using this result along with the tight I/O bounds on individual tensor contractions, we explore I/O lower bounds for various other fusion choices. We conclude that the best way to fuse a four-index transform is to either fuse in pairs of two, or to fuse all four contractions together.

### 4.4.1 Necessary condition: utility of fusion

Consider the four-index transform re-written as shown in Equation 4.2. We can identify the necessary condition for utility of fusion by applying Lemma 4.3.3 and Corollary 4.3.4 to the first two contractions of the four-index transform.

Let $n_\alpha, n_\beta, \ldots$ be the size of the indices along $\alpha, \beta, \ldots$. In practice, $n_\alpha = n_\beta = n_k = \ldots$. To simplify the notation (the following reasoning also remains valid if they are different but within the same order of magnitude), let us consider a common value of $n$.

Let us recall for convenience the first two contractions in the 4-index transform.

$$O1[\alpha,j,k,l] = \sum_i A[i,j,k,l].B[\alpha,i] \quad (4.4)$$

$$O2[\alpha,\beta,k,l] = \sum_j O1[\alpha,j,k,l].B[\beta,j]$$

²$I1$ represents the input set of the CDAG for $op1$, $O2$ the output set of the CDAG for $op2$
Listing 4.5: Tensor contraction schedule that achieves I/O equal to sum of its input and outputs

```c
//Does I/O equal to |C|+|A|+|B| if S >= n^2 + n + 1
load B[alpha*,i*] //requires size n^2
for l, k, j
    copy (A_buf[i*], A[i*,j,k,l]) //requires size n
for alpha
    scalar c //allocate a scalar
    for i
        c += A_buf[i] * B[alpha,i]
C[alpha,j,k,l] += c
```

Each tensor contraction can be represented as a matrix multiplication (seeing resp. \((j,k,l) / (\alpha,k,l)\) as a unique dimension in \(A / O1\)). Using this representation, we compute the lower bound on I/O for each of the contractions for a cache size \(S\). Applying the known (tight) lower bound for matrix-matrix multiplication (e.g., [40]), the lower bound for the first two contractions \(IO_{opt}(op1)\) and \(IO_{opt}(op2)\) is given by:

\[
IO_{opt}(op1) = IO_{opt}(op2) = \Theta\left(\frac{n^5}{\sqrt{S}}\right)
\]

Using Corollary 4.3.4, we get the following condition for fusion to be useful:

\[
\Theta\left(\frac{n^5}{\sqrt{S}}\right) = IO_{opt}(op1) + IO_{opt}(op2) = \mathcal{O}\left(n^4\right)
\]

In other words, \(S = \Omega\left(n^2\right)\). This condition must hold for fusion to be useful for any two consecutive contractions in the four-index transform.

### 4.4.2 Tight I/O bound for useful fusion

In this subsection, we identify a tight I/O bound for fusion of two consecutive contractions in the four-index transform when \(S = \Omega\left(n^2\right)\). Our results are summarized in the following theorem.

**Theorem 4.4.1** (usefulness of fusing 2-contractions). Consider any two consecutive pair of tensor contractions \(op1\) and \(op2\) in four-index transform, where the size of all
dimensions is the same order of magnitude, say, \( n \). Let \( A/O2 \) be the input/output of \( op1/op2 \). Then,

\[
\text{fusion is useful only if } S = \Omega(n^2).
\]

In that case, the tight lower bound is given by \( IO_{\text{opt}}(op1 \cup op2) = |A| + |O2| \).

Proof. Without loss of generality let us use the first two contractions in the four-index transform (see Eq. 4.4) to represent any pair of consecutive contractions. We have already shown that fusion for a pair of consecutive tensor contractions in the four-index transform is useful only if \( S = \Omega(n^2) \). Here, we prove the latter part of the theorem. This is done in two steps. First, we show that \( |A| + |O2| \) is a lower bound on I/O for a fused schedule when \( S = \Omega(n^2) \). Next, we show that we can construct a fused schedule that achieves this lower bound.

Note that if \( S = \Omega(n^2) \), then a tight I/O lower bound on individual tensor contractions in the four-index transform is given by the sum of the sizes of the input and output tensors. As each input tensors must be read at least once and the output tensor must be written at least once, this is a lower bound to the I/O of the tensor contraction. Lst. 4.5 achieves this I/O. Therefore, it is a tight bound. Now using Lemma 4.3.3, the lower bound on I/O for a fused schedule is given by:

\[
(|A| + |O1|) + (|O1| + |O2|) - 2|O1| = |A| + |O2| \tag{4.7}
\]

Additionally, when \( S = \Omega(n^2) \), a schedule can be constructed that achieves this bound. Loops \( k \) and \( l \) can be fused: for any fixed value of \( k \) and \( l \), the \( n^2 \) values of \( A[i,j,k,l] \) can be initially loaded, allowing I/O-free computation of the \( n^2 \) values of \( O1[\alpha,j,k,l] \). This, in turn, allows I/O-free computation of the \( n^2 \) values of \( O2[\alpha,\beta,k,l] \), which are then stored. The I/O for such a schedule (shown in Lst. 4.6)
Listing 4.6: Fused pair of tensor contractions that achieves I/O equal to sum of size of its input and outputs

```c
//if S >= 3n^2 + n + 1
//Total I/O of |C|+|B0|+|B1|+|A|
//requires 2n^2 memory
load B1[alpha*,i*], B2[beta*,j*]
for l, k
malloc (I1_buf, n^2)
for j
malloc (A_buf, n)
copy (A_buf, A[i*,j,k,l])
for alpha, i
I1_buf[alpha,j] += A_buf[i] * B1[alpha,i]
for beta, alpha, j
C[alpha,beta,k,l] += I1_buf[alpha,j]*B2[beta,j]
```

is exactly the size of inputs and outputs (there is no intermediate I/O in executing the operations). Therefore, |A| + |O2| is a tight I/O bound.

4.4.3 Fusion Choices: four-index transform

The optimal fusion strategy is one that minimizes the I/O between slow and fast memory. Different unique ways to fuse four-index transform are as follows: 1. No fusion; 2. Fuse first two and last two; 3. Fuse only a single pair; 4. Fuse three contractions; 5. Fuse all contractions. This section’s goal, as summarized in the following frame, is to identify which strategy is the best, i.e., the one with the least I/O.

| Given: | S = \Omega(n^2). |
| Problem: | What fusion among op1, op2, op3, and op4 is best? |
| Result: | Let op12 = op1 \cup op2, op12/23 = op12 + op23, op123 = op1 \cup op2 \cup op3, etc. IO_{opt}(op1234) \leq IO_{opt}(op12/34) < IO_{opt}(op123/4) |

The I/O lower bound for each of the fusion choices can be computed using Lemma 4.3.3, in conjunction with the tight I/O bound for individual tensor contractions and the result from Theorem 4.4.1. We already know that the tight I/O
bound for each individual (or pair of) contractions is the sum of sizes of its input and output. For fusion of three or more contractions, the sum of its input and output sizes is a correct lower bound but not necessarily a tight bound. Below, we summarize the I/O bounds for each fusion choice.

We use the following notation: \( op_{1234} \) (resp., \( op_{12/34} \), \( op_{123/4} \), etc.) denotes fused computation of all tensor contractions (respectively, first two and last two with forced spilling in between, first three, etc.). Thus (using the sets as in Equation 4.2):

\[
\begin{align*}
IO_{opt}(op_{123/4}) &= |A| + |O_1| + |O_1| + |O_2| + |O_2| + |O_3| + |O_3| + |C| \\
IO_{opt}(op_{12/34}) &= |A| + |O_2| + |O_2| + |C| \\
IO_{opt}(op_{123/4}) &= |A| + |O_1| + |O_1| + |O_3| + |O_3| + |C| \\
IO_{opt}(op_{1234}) &\geq |A| + |O_3| + |O_3| + |C| \\
IO_{opt}(op_{1234}) &\geq |A| + |C|
\end{align*}
\]

Thus far, we have assumed that the size of all 4D tensors is \( n^4 \), thus \( IO_{opt}(op_{123/4}) \geq IO_{opt}(op_{1234}) \). However, in reality, \( IO_{opt}(op_{123/4}) > IO_{opt}(op_{1234}) \) because \( |O_3| \) is bigger than \( |O_2| \) due to symmetry (see Table 4.1). In addition, we also have trivially that \( IO_{opt}(op_{1234}) \leq IO_{opt}(op_{1234}) \). Thus, we get the following total order.

**Theorem 4.4.2** (order of fusions). Consider four consecutive tensor contractions \( op_1, op_2, op_3, \) and \( op_4 \) and the previously defined notation for fusion. Let \( S = \Omega(n^2) \). Then,

\[
IO_{opt}(op_{1234}) \leq IO_{opt}(op_{1234}) < IO_{opt}(op_{1234})
\]

In other words, the lower bounds show that fusion of the first two and last two has lower I/O than no fusion, while fusing three contractions does not lower the I/O
further. The only way it may be possible to reduce the I/O further is to fuse all four contractions together to achieve \(|A|+|C|\) via full reuse of intermediates.

### 4.5 Necessary and sufficient conditions for full reuse

In this section, we investigate the necessary and sufficient conditions for full reuse of all intermediates, resulting in an I/O of \(|A|+|C|\). Note that if the size of fast memory (S) is larger than the sum of all the intermediates in the four-index transform, then achieving an I/O of \(|A|+|C|\) is trivial. Even a schedule that performs each of the tensor contractions individually will achieve this I/O. However, when the intermediates are too big to fit in fast memory, is full reuse of intermediates still possible? If so, under what conditions?

We claim that, \(S \geq |C|\) is a necessary and sufficient condition for a tight I/O bound of \((|A|+|C|)\). The results from this section are summarized in the following frame.

| Given: Four-index transform \(op_{1234}\) as in Equation 4.2 |
| Problem: What are necessary and sufficient conditions on S such that full-reuse is possible? |
| Thm: \(IO_{opt}(op_{1234}) = |A|+|C|\) iff \(S \geq |C|\) |

#### 4.5.1 Necessary conditions

In this subsection, we show that if \(IO_{opt}(op_{1234}) = |A|+|C|\) then \(S \geq |C|\). We first consider the case without symmetries and then show that the proof holds even with symmetries.

**Theorem 4.5.1 (No symmetry).** Let \(op_{1234}\) be the CDAG corresponding to the computation described by Equation 4.2.
Then, $IO_{opt}(op1234) = |A|+|C|$

only if $S \geq \min (|A|, |O1|, |O2|, |O3|, |C|)$

Proof. To achieve the I/O lower bound $(|A|+|C|)$, the intermediates have to be fully reused without any intervening I/O in between the operations. Let $C[\alpha, \beta, \gamma, \delta]$ be the first computed element of the output, and the corresponding time stamp right before its computation be $T_f$. We call live set (also denoted $W$), the already loaded values of $A$, partially or fully computed values of $O1$, $O2$, $O3$ and $C$ that are to be used later on. We argue that this live set is at least of size $|C|$. Let $F0$, $F1$, $F2$, $F3$ and $F4$ be the already loaded/computed values of $A$, $O1$, $O3$ and $C$, respectively that may or may not be used later on. Let $D0, \ldots, D4$, be the set of values in $F0, \ldots, F4$, respectively, that will not be used later on. Therefore, the size $|W|$ of the live set is given by

$$(|F0|−|D0|) + (|F1|−|D1|) + (|F2|−|D2|) + (|F3|−|D3|) + (|F4|−|D4|)$$

Note that to be spill-free (i.e., without any intervening I/O in between contractions), a schedule must be such that $|W| \leq S$. Indeed, $W$ corresponds to simultaneously live variables: if all of them cannot fit in collective memory, it means some need to be spilled. We now establish a lower-bound on $|W|$.

Note that each element of $C$ depends on all values of tensor $A$, and at time step $T_f$ none of the output elements are yet fully computed. Therefore, at time step $T_f$, $F0 = A$ and $D4 = \emptyset$. $|W|$ simplifies to

$$(|A|−|D0|) + (|F1|−|D1|) + (|F2|−|D2|) + (|F3|−|D3|) + (|F4|)$$

(4.8)

Let us recall the computation of the first tensor contraction:

$$O1[\alpha, j, k, l] = A[i, j, k, l] * B0[\alpha, i]$$
There are two key observations:

1. If a given $A[i,j,k,l]$ is in $D_0$, then all $O_1[*j,k,l]$ must be in $F_1$;
2. If $D_{0,j,k,l}$ denotes all the $A[*j,k,l]$ (at most $n_i$ elements) in $D_0$ and $F_{1,j,k,l}$ denotes all the $O_1[*j,k,l]$ (exactly $n_\alpha$ elements) in $F_1$, then $|F_{1,j,k,l}| / |D_{0,j,k,l}| \geq n_\alpha / n_i$.

Thus, $|F_1| \geq |D_0| \times n_\alpha / n_i = |D_0| \times |O_1| / |A|$. Applying the same reasoning on the remaining contractions leads to the following lower bound for $|W|$: 

$$|W| \geq |A| - |D_0| + (|D_0| \times |O_1| / |A| - |D_1|) + (|D_1| \times |O_2| / |O_1| - |D_2|) + (|D_2| \times |O_3| / |O_2| - |D_3|) + (|D_3| \times |C| / |O_3|)$$

which can be rewritten as:

$$|W| \geq |A| + |D_0| \times (|O_1| / |A| - 1) + |D_1| \times (|O_2| / |O_1| - 1) + |D_2| \times (|O_3| / |O_2| - 1) + |D_3| \times (|C| / |O_3| - 1)$$

which itself (as $|D_0| \leq |A|$, $|D_1| \leq |O_1|$, etc.) can be bounded by

$$|W| \geq \min(|A|, |O_1|, |O_2|, |O_3|, |C|)$$

The tensors in the four-index transform contains permutation symmetry that can be represented as $A[ij,kl]$, $O_1[\alpha,j,kl]$, $O_2[\alpha\beta,kl]$, $O_3[\alpha\beta,\gamma,l]$, and $C[\alpha\beta,\gamma\delta]$. While all tensors contain some degree of permutation symmetry, the output tensor $C$ also contains spatial symmetry. Because this makes the output $|C|$ the smallest of all five tensors, we expect the necessary condition for full reuse to be possible to be $S \geq |C|$.

**Theorem 4.5.2 (With Symmetry).** Consider the four-index transform computation $op_{1234}$ (Equation 4.2 with symmetries in Table 4.1). Then,
Proof. We note that Equation 4.8 gives the live-set for both symmetric and non-symmetric contractions. We then prove by contradiction that Equation 4.8 is greater than \( |C| \). To make Equation 4.8 smaller than \( |C| \), each term in Equation 4.8 must be smaller than \( |C| \). This forces \( |D0| \) to be greater or equal to \( |A| - |C| \), i.e., \( |D0| \geq 7|A|/8 \) based on Tab. 4.1. Similar to the previous proof, we have that \( D0_{j,kl} \) has at most \( j \) elements (exploiting the symmetry \( j < i \)) and that \( F1_{j,kl} \) has exactly \( n_\alpha \) elements. Again, this leads to \( |F1| \geq |D0| \times |O1|/|A| \) i.e., \( |F1| \geq 7/8|O1| \geq |C| \). In general, enforcing the condition that each term must be smaller than \( |C| \) forces \( Li \) on the next term to be much larger than \( |C| \) for all terms except the last one, where \( |F4| \) is forced to be \( |C| \). This leads to a contradiction, proving that \( S \geq |W| \geq |C| \). \( \square \)

We can derive a similar result for the fusion of the first three tensor contractions. Indeed, following the same reasoning, considering instead the time stamp when the first output of \( O3 \) is computed, we can replace \( |A| \) in Equation 4.8 by \( |F0| = n^3 \). Assuming \( |D0| \ll |F0| \) leads to \( |W| \geq \Omega(n^3) \). Assuming \( |D0| = \Omega(n^3) \) leads to \( |F1| = \Omega(n^3) \). Repeating the same analysis for \( |D1|, |F2| \), etc. proves that \( |W| \geq \Omega(n^3) \) is a necessary condition for the first three tensor contractions to be spill free.

**Theorem 4.5.3 (Three Contractions).** Consider the four-index transform computation. Let \( op123 \) be the fusion of the first three contractions. Then,

\[
IO_{opt}(op123) = |A| + |O3| \text{ only if } S = \Omega(n^3)
\]
Listing 4.7: Fused four-index transform that achieves I/O equal to the size of input and output

```c
/* In this code a=alpha,b=beta, c=gamma,d=delta. If S > |C|,
then total I/O is
|C|+|B1|+|B2|+|B3|+|B4|+|A| */
for l
malloc (O1,n^3); malloc(A_buf,n*3)
copy (A_buf, A[(i>j),k*,l])
for a, i, j, k
O1[a,j,k] += A_buf[i>j,k]*B1[a,i]
delete(A_buf); malloc(O2,n^3/2)
for a > b, j, k
O2[a>b,k] += O1[a,j,k]*B2[b,j]
delete (O1); malloc(O3, n^3/2)
for a>b, c, k
O3[a>b,c] += O2[a>b,k]*B3[c,k]
delete (O2)
for a>b, c>d
C[a>b,c>d] += O3[a>b,c]*B4[d,l]
```

4.5.2 Sufficient condition

Lst. 4.7 shows a fused schedule op1234, such that \( IO_{\text{opt}}(\text{op1234}) = |A|+|C| \), when \( S \geq |C| \), thus establishing that \(|A|+|C|\) is a tight I/O bound on op1234. Keeping in mind that \( S \geq |C| \) must be true, the key idea we used to obtain this schedule was to keep the entire output \( C \) in fast memory throughout the entire computation.

In this schedule, we fuse loop 1 across all contractions. Then for each iteration of loop 1, we compute \( O1[*,*,*,l] \), \( O2[*,*,*,l] \), and \( O3[*,*,*,l] \). Each iteration of 1, thus requires \( A[*,*,*] \) as the input, and no two iteration of 1, depends on the same values of \( A \), \( O1 \), \( O2 \), or \( O3 \). Thus values of the input and the intermediates for a particular iteration of 1 can be discarded before the next iteration as it is fully reused during the iteration. However, each iteration makes partial contribution to the entire output tensor \( C \). Therefore, \( C \) must be kept in fast memory during all iterations of 1.

This schedule achieves an I/O of \(|A|+|C|\) if \( S \geq |C| + 2n^3 \), where the first term corresponds to size of fast memory that allows \( C \) to be kept in fast memory across
Listing 4.8: Implementation that fuses all four contractions to allow execution of the
largest possible problem size without recomputation

```c
int node_id = GA_Nodeid()
for l
    GA_Create(A,n^3/2);
    for i>j, k
        bufA[i>j,k] = ComputeA(i>j,k,l)
        GA_Put(bufA,A[i>j,k])
    GA_Create(O1,tilewidth* n^3)
    for j, k
        if (node_id owns O1[alpha*,j,k])
            bufA = GA_Get(A[i*,j,k,l])
            for alpha
                for i
                    bufB = ComputeB(alpha,i)
                    DGEMM(bufA[i],bufB,bufO1)
                GA_Put(bufO1, O1[alpha,j,k])
            GA_Sync()
    GA_Create(O2,tilewidth*n^3/2)
    for alpha, k
        if (node_id owns O2[alpha,beta*,k])
            bufO1 = GA_Get(O1[alpha,j*,k])
            for beta //less than alpha
                for j
                    bufB = ComputeB(beta,j)
                    DGEMM(bufO1[j],bufB,bufO2)
                GA_Put(bufO2, O2[alpha>beta,k])
            GA_Sync(); delete O1
    GA_Create(O3,tilewidth*n^3/2)
    for alpha>beta
        if (node_id owns O3[alpha>beta,gamma*])
            bufO2 = GA_Get(O2[alpha,beta,k*,l])
            for gamma
                for k
                    bufB = ComputeB(gamma,k)
                    DGEMM(bufO2[gamma],bufB,bufO3)
                GA_Put(bufO3, O3[alpha>beta,gamma])
            GA_Sync(); delete O2
        for alpha>beta, gamma
            if (node_id owns C[alpha>beta,gamma>delta])
                bufO3 = GA_Get(O3[alpha>beta,gamma])
                for delta //less than gamma
                    bufB = Compute(delta,1)
                    DGEMM(bufO3,bufB,bufC)
                GA_Acc(bufC,C[alpha>beta,gamma>delta])
            GA_Sync(); delete O3
```

iterations of l, and the second term keeps the intermediates from spilling. Thus, we
have a schedule op1234 such that \(IO_{opt}(op1234) = |A|+|C|\), when \(S \geq |C|\) (as \(n^3\) is
negligible compared to \(|C|\)).
4.6 Improving the Four-index transform in NWChem

Based on our analysis of fusion in four-index transform, we develop a fused schedule for the four-index transform in NWChem that has the following properties:

1. Largest zero-spill four-index transform: For a given distributed memroy system, our schedule allows for the execution of the largest possible four-index transform problem, without spilling into disk, and recomputation.

2. Minimization of communication volume: Our fused schedule minimizes the communication volume between distributed memory nodes, thus enhancing performance for systems where communication cost is the performance bottleneck.

4.6.1 Largest zero-spill four-index transform

Achieved Communication volume (global memory–disk)

\[ 0 \] \hspace{1cm} (4.9)

Memory requirement

\[
\frac{N_i \times N_j \times N_k \times T_l}{2} + \frac{N_\alpha \times N_\beta \times N_k \times T_l}{2} + \frac{N_\alpha \times N_\beta \times N_\gamma \times N_\delta}{32} \] \hspace{1cm} (4.10)

In a distributed-memory system, we consider the disk as the unbounded slow memory, and the global memory as the fast memory, then from Theorem 4.5.2, size of global memory must be larger than \(|C|\), to achieve an I/O of \(|A| + |C|\). Our implementation of the four-index transform is based on the Lst. 4.7, which achieves this I/O when the size of global memory is greater than \(|C|\). Because \(S \geq |C|\) is a necessary condition for complete reuse of the intermediates, it is guaranteed that our solution will allow spill-free execution of the largest possible four-index transform for a given amount of global memory.
In addition, the I/O for our implementation between disk and global memory is actually zero. The I/O reduces from $|A| + |C|$ to zero because of two reasons:

1. The input tensor is produced on the fly, i.e., elements of it can be cheaply computed without ever having to store the entire tensor. It is produced in fast memory as needed.

2. For all problems of practical interest, $S \geq |C|$ The output $C$ fits in fast memory, so it can be consumed by the next step in the calculation, without ever storing it in disk.

Lst. 4.8 shows the details of the implementation. Each loop iterates over tile indices. Loop $l$ (line 2) is the fused loop over tile $l$. Iterations of this loop are executed sequentially, while the computation within this loop is parallelized. Each iteration of loop $l$ computes an intermediate sub-tensor of size $N^3$. In other words, for a fixed value of $l$, a slice of $N^3$ elements of $O_1$, $O_2$, and $O_3$ are produced. For example lines 3..14 show the computation of a sub-tensor of $O_1$. The sub-tensor is distributed and requires $O(N^3)$ global memory. Each intermediate sub-tensor is used to compute the next intermediate sub-tensor. This computation is parallelized. Each processor only computes the tiles it owns. Finally, the intermediate sub-tensor $O_3$ is used to compute partial contributions to output tensor $C$. Each iteration of loop $l$ computes a partial contribution to the entire output tensor $C$. Therefore, $C$ must fit in global memory.

4.6.2 Minimizing communication volume

Achieved Communication volume (global memory–local memory)

$$|A| + |O_2| + |O_2| + |C|$$ (4.11)
Listing 4.9: Fused schedule op12/34

```c
// if S > n^2, total I/O = |A|+|O2|+|O2|+|C|
for k > l
  for alpha
    for beta // less than alpha
      for j
  for alpha>beta
    for gamma
      O3[l] = O2[alpha>beta,k>l] * B3[gamma,k]
      for delta // less than gamma
        for l
          C[alpha>beta,gamma>delta] +=
          O3[gamma,l]*B4[delta,l]
```

Memory requirement

\[
\frac{N_i \cdot N_j \cdot N_k \cdot T_l}{2} + \frac{N_\alpha \cdot N_j \cdot N_k \cdot T_l}{2} + \frac{N_\alpha \cdot N_\beta \cdot N_k \cdot T_l}{2} + \frac{N_\alpha \cdot N_\beta \cdot N_\gamma \cdot T_l}{2} + \frac{N_\alpha \cdot N_\beta \cdot N_\gamma \cdot N_\delta}{32}
\]  

(4.12)

I/O at the second level of the memory hierarchy—between the global (distributed) memory and the local memory—corresponds to communication between distributed memory nodes. At this level of memory hierarchy, global memory is the unbounded slow memory, while local memory is the bounded fast memory. We can minimize the I/O between global memory and local memory by making the following two observations:

1. The computation within the fused loop in Lst. 4.8 is still a four-index transformation (referred to as the inner four-index transform). By fusing loop \( l \), we have reduced the size of index \( l \) within the fused loop to either 1, in absence of tiling, or \( T_l \) in the presence of tiling, where \( T_l \) is the tile width. However, the fusion analysis presented in this paper still applies to the inner four-index transform.

2. For large problems of interest, the size of the local memory is smaller than the final
output $C$ of the inner four-index transform. Thus, full reuse of intermediates is not possible based on Theorem 4.5.2. However, from Theorem 4.5.2 and Theorem 4.4.2, we know that schedule op12/34 (shown in Lst. 4.9) achieves an I/O bound lower than all other fusion choices, when $S = \Omega \left( \left( \frac{1}{\alpha} n^2 \right) \right)$ and $S < |C|$. Thus, op12/34 can be used to minimize communication volume for the inner four-index transform.

Such an implementation is shown in Lst. 4.10. In addition to fusing loop $l$, this implementation performs additional fusion of the first two and the last two contractions of the inner four-index transform. $k$ and $\alpha$ loops are fused between the first two contractions, and $\alpha$, $\beta$ and $\gamma$ loops are fused between the second two contractions, similar to the op12/34 schedule shown in Lst. 4.9.

4.6.3 Mapping to processors

On a conventional memory hierarchy with a single slow and fast memory, op12/34 eliminates I/O corresponding to tensors $O_1$ and $O_3$. To achieve the same effect on a parallel system with multiple fast memory (corresponding to local memory of individual processors), mapping of computation to processors must be chosen carefully.

Consider the fused schedule for the first two inner contractions in Lst. 4.10. Inside the fused loop $k$, each element of $A$ is reused to compute multiple elements of $O_1$ along $\alpha$, and each element of $O_1$ is reused to compute multiple elements of $O_2$ along $\beta$. If different values of $\alpha$ (or $\beta$) are mapped to different processors, then same values of $A$ (or $O_1$) must either be computed at each of the processor, or it must be communicated. To avoid, both communication and redundant computation of $A$, or $O_1$, all values of $\alpha$ and $\beta$ for a given $k$ and $l$ must be computed on the same processor. As a result, only the fused loop $k$ can be parallelized. Similarly, for
Listing 4.10: Implementation that performs outer fusion to maximize problem size and inner fusion to minimize communication.

```c
int node_id = GA_Nodeid() //this process’s rank
for l
    GA_Create(A,n^3/2)
    for (i>j,k)
        bufA[i>j,k] = ComputeA(i>j,k,l)
        GA_Put(bufA,A[i>j,k])
    GA_Create(O2,tilewidth* n^3/2)
    for k
        if (node_id owns O2[(alpha>beta)*,k])
            bufA = GA_Get(A[(i>j)*,k])
        for alpha
            for j, i
                bufB = ComputeB(alpha,i)
                DGEMM(bufA[i>j],bufB,bufO1[j])
        for beta //less than alpha
            for j
                bufB = ComputeB(beta,j)
                DGEMM(bufO1[j],bufB,bufO2[beta])
            GA_Put(bufO2[beta], O2[alpha>beta,k])
    GA_Sync()
for alpha > beta
    if (node_id owns C[alpha>beta,(gamma>delta)*])
        bufO2 = GA_Get(O2[alpha>beta,k*])
    for gamma, k
        bufB = ComputeB(gamma,k)
        DGEMM(bufO2[k],bufB,bufO3)
    for delta //less than gamma
        bufB = ComputeB(delta,l)
        DGEMM(bufO3,bufB,bufC)
    GA_Acc(bufC,C[alpha>beta,gamma>delta])
GA_Sync(); delete O2
```

the last two contractions, only the fused loops \(alpha\) and \(beta\) can be parallelized.

**Parallelism vs communication+load imbalance:** For large numbers of processors, there may not be enough parallelism in the inner fused schedule of the first two contractions as only loop \(k\) can be parallelized. We overcome this problem in our implementation by allowing parallelization of \(alpha\), which results in increased communication for \(A\), by a factor equal to the parallelization of \(alpha\). As this is also true for the unfused inner version, there is still a significant reduction in communication volume due to elimination of data movement for \(O1\) and \(O3\).
Another side effect of parallelization of alpha is a potential for load imbalance. The load imbalance is due to permutation symmetry between alpha and beta. As alpha $\geq$ beta, a processor computing alpha = 2 only computes for two values of beta, while a processor computing for alpha = 3 computes three values of beta. Thus, there is potential for load imbalance. There are alternative load balancing strategies or strategies to increase parallelism, such as block cyclic-distribution of tensors, or nested tiling of l, allowing loop k and parts of loop l to be parallelized. However, those solutions are beyond the scope of this paper, as they are orthogonal to our fusion analysis.

4.6.4 Space-time trade-off due to symmetry breaking

In the absence of any permutation symmetry, the fused four-index transform and the unfused one have the same number of operations (additions and multiplications). However, in the presence of permutation symmetry, fusing loop 1 across all four contraction in the four-index transform either requires breaking the symmetry between k and l indices, which doubles the number of computations for producing $O_1$ and $O_2$, or it requires more fast memory to avoid spills.

To see this, consider the second and third contractions in the four-index transform given by:

$$O_2[\alpha, \beta, kl] = \sum_j O_1[\alpha, j, kl].B_2[\beta, j]$$

$$O_3[\alpha, \beta, \gamma, l] = \sum_k O_2[\alpha, \beta, kl].B_3[\gamma, k]$$

In the absence of fusion, we can first compute $O_2[\alpha \geq \beta, k \geq l]$. Then, during computation of $O_3$, if we need a $O_2[\alpha \geq \beta, k, l]$ where $k < l$, we can get it from $O_2[\alpha \geq$
Table 4.2: Execution time (seconds) of four-index transform for tce_hyperpolar_ccsd_big on RI

<table>
<thead>
<tr>
<th>Cores</th>
<th>fused outer</th>
<th>fused outer+inner</th>
<th>NWChem Best</th>
<th>unfused</th>
<th>recompute</th>
<th>12-34 fused</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1589</td>
<td>1632</td>
<td>4925</td>
<td>Fail</td>
<td>4925</td>
<td>Fail</td>
<td>3.10</td>
</tr>
<tr>
<td>64</td>
<td>915</td>
<td>978</td>
<td>2273</td>
<td>Fail</td>
<td>2273</td>
<td>Fail</td>
<td>2.45</td>
</tr>
<tr>
<td>128</td>
<td>523</td>
<td>599</td>
<td>350</td>
<td>350</td>
<td>1147</td>
<td>1055</td>
<td>0.66</td>
</tr>
</tbody>
</table>

$\beta, l, k$ because $O2[\alpha \geq \beta, k, l] = O2[\alpha \geq \beta, l, k]$ due to permutation symmetry. Thus, we only compute $n^4/4$ elements of $O2$, where $n$ is the size of each dimension of $O2$.

Now, consider the number of operations performed in the presence of our fused schedule presented in Lst. 4.7, which fuses loop 1 to obtain a schedule with I/O $|A| + |C|$ if $S > |C|$. In this schedule, loop 1 is fused across all contractions. During each iteration of 1, we compute $O2$ and $O3$ for the corresponding value of 1. Let us assume that we are computing for $1 = 5$. To compute $O3[\alpha \beta, \gamma, 5]$, we need to compute $O2[\alpha \geq \beta, *, 5]$ during iteration 1=5. This requires breaking permutation symmetry between $k$ and $l$.

Symmetry breaking means that non-unique values of a symmetric tensor will be produced multiple times. As a concrete example, $O2[\alpha \geq \beta, 2, 5]$ is required to compute $O3[\alpha \beta, \gamma, 5]$ and $O2[\alpha \geq \beta, 5, 2]$ is required to compute $O3[\alpha \beta, \gamma, 2]$. Thus, they will be computed at $l = 2$ and $l = 5$, respectively. However, both of these values are the same due to permutation symmetry. In other words, breaking permutation symmetry results in doubling the number of elements of $O2$ that are produced from $n^4/4$ to $n^4/2$. 

114
Table 4.3: Execution time (seconds) of four-index transform for Uracil using cc-pVQZ basis-set on RI

<table>
<thead>
<tr>
<th>Cores</th>
<th>fused outer</th>
<th>fused outer+inner</th>
<th>NWChem Best</th>
<th>unfused</th>
<th>recompute</th>
<th>12-34 fused</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>2826</td>
<td>4252</td>
<td>Fail</td>
<td>Fail</td>
<td>Fail</td>
<td>fail</td>
</tr>
</tbody>
</table>

Table 4.4: Execution time (seconds) of four-index transform for Uracil using cc-pVQZ basis-set on Cascade

<table>
<thead>
<tr>
<th>Cores</th>
<th>fused outer</th>
<th>fused outer+inner</th>
<th>NWChem Best</th>
<th>unfused</th>
<th>recompute</th>
<th>12-34 fused</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>3160</td>
<td>2470</td>
<td>14,566</td>
<td>Fail</td>
<td>Did not finish</td>
<td>14,566</td>
<td>5.89</td>
</tr>
<tr>
<td>1024</td>
<td>2690</td>
<td>1692</td>
<td>6296</td>
<td>Fail</td>
<td>Did not finish</td>
<td>6296</td>
<td>3.72</td>
</tr>
</tbody>
</table>

On the other hand, trying to preserve the permutation symmetry increases the requirement on the fast memory. For $l=5$, we should compute $O2[\alpha \geq \beta, k \geq 5]$, if we want to compute only $n^4/4$ elements of $O2$. Any value $O2[\alpha \geq \beta, k, 5]$ where $k < l$ must be therefore obtained from $O2[\alpha \geq \beta, 5, k]$. In the fused code, $O2[\alpha \geq \beta, 5, k]$ is produced when $l = k$. This creates dependencies between different loop iterations of $l$, i.e., values produced at one iteration of $l$ is consumed by multiple iterations of $l$, thus requiring it to be stored after the iteration in which it is produced. On average, each iteration of $l$ requires values from $n/2$ other iterations of $l$ because on average $n/2$ values of $k$ is greater than $l$. Without going into detail, this requires approximately $|O2|/2$ elements to be stored for later reuse, increasing the size of $S$ required for a spill-free computation of the four-index transform. This is equivalent to not doing fusion.
In summary, while our fused schedule is still an \( n^5 \) algorithm, the total computation is increased. It is approximately \( 1.5 \times \) the original computation because the computation on the first and second contractions is doubled. However, this is a small price to pay to get a tight lower bound on I/O of \(|A| + |C|\) when \( S > |C| \). However, if \( S \) is greater than the size of the intermediates, we should use a fully unfused solution, because it will achieve an I/O of \(|A| + |C|\) without any additional computation.

4.7 Experimental Results

We show that for problems that run at the limit of available memory size, where the amount of available global memory is less than the size of the intermediates, our fused implementations significantly outperforms the best NWChem implementation by up to a factor of 6. In addition, we show that our fused implementation allows running large problem sizes on small systems that are just not possible with the current NWChem implementation. For smaller problems, NWChem’s unfused implementation is approximately 50\% faster than our fused implementation due to absence of symmetry breaking and full reuse of intermediates.

We ran our experiments on two computing clusters: i) RI is a small infiniband cluster where each node of has two 4-core 2.53 GHz Intel Xeon E5630 (Westmere) processors and 24 GB main memory per node, connected as a full fat tree by QDR Infiniband (40 Gbps). ii) Cascade FDR Infiniband (14Gbps) large supercomputer with 1440 2.6GHz dual-socket Intel Xeon E5-2670 processors (8 cores per socket) with 128GB of main memory. We used NWChem 6.5 compiled with GCC 4.4.7 on RI and NWChem6.5 compiled with Intel ICC Compiler on Cascade.
Medium problem. On RI, we ran a medium-size problem—hyperpolar_ccsd_big. The problem, with size determined in terms of the number of orbitals, has 368 alpha and beta orbitals, for a total of 736 orbitals. Table 4.2 shows execution times and speedups of our fastest fused implementation over the fastest NWChem implementation. Table 4.2 shows that when intermediates do not fit in collective memory, fused : outer is significantly faster than the best NWChem implementation. For up to 64 cores, the total collective global memory is not large enough to store the intermediates, and we were only able to run recompute. On 128 nodes, we were able to run the unfused version that fully reuses intermediates, achieving zero-spill without symmetry breaking.

Large problem. We ran the four-index transform on a Uracil molecule using the cc-PVQZ basis-set. This problem has 698 alpha and beta orbitals for a total of 1396 orbitals. Table 4.3 shows the execution time for the fused implementation on RI, and Table 4.4 shows the execution time and speedup over the best NWChem algorithm on Cascade.

On RI, none of the NWChem implementations had enough resources to run Uracil. This shows that our fused implementation allows running much larger problem on a given system that currently possible with state-of-art implementation in NWChem. On Cascade, with 5 times more per-node memory than RI, we were able to compute the four-index transform using 12-34 fused and compare it with our fused implementation. Table 4.4 shows significant speed up from our fused algorithm for this problem size.

Note that despite 5 times smaller memory in RI, we were still able to run this large problem size with ease. On paper, both RI and Cascade are quite evenly matched in
terms of hardware, except for the size of available memory. The performance results for 512 cores are also comparable between the two systems. This is very encouraging, as it proves that with the new fused algorithm we can run much larger problems with the given amount of global memory than before, without performance penalties.

**Performance trade-off.** Notice that fused:outer+inner provides additional performance on Cascade but not on RI. As RI is a full fat tree with 40Gbps infiniband, communication is not a performance bottleneck. Thus, the load imbalance introduced by fused:outer+inner decreases the performance. However, on Cascade, communication is the primary bottleneck. Thus, despite the load imbalance, fused:outer+inner affords additional performance.

### 4.8 Conclusions

In this paper, we have addressed the development of a high-performance parallel implementation of the four-index integral transformation, a compute-intensive calculation used in many quantum chemistry models. The optimization of data movement overheads is feasible through loop fusion and tiling of the nested loop computations for the sequence of tensor contractions used in an operation-minimal form of the computation. However, the number of possible fusion configurations and tile size choices makes the development of an optimized implementation challenging. Several different implementations have been developed over the last two decades for the four-index transformation in NWChem, successively attempting to improve on previous versions.

In this paper, we pursued a radically different approach to addressing the problem of determining which of many fusion choices we should consider. Instead of the conventional approach of explicitly modeling different possible fused loop configurations...
and modeling their expected performance, we instead used lower bounds modeling to characterize conditions where fusion might have utility and where it is futile because the maximal possible improvement from fusion is much lower than the minimal data movement overheads for each of the unfused components. The lower-bounds-based analysis enabled the development of an efficient parallel implementation that was demonstrated to be significantly better than the best current option in the production NWChem code. The newly developed implementation of the four-index transform is slated for incorporation into NWChem in future releases.
Chapter 5: Framework for Optimizing CNNs on CPUs

5.1 Introduction

Convolutional Neural Networks (CNN) are a class of Artificial Neural Networks (ANN) that are highly efficient at the pattern recognition tasks that underlie difficult AI problems in a variety of domains such as speech recognition, object recognition, and natural language processing [83, 116, 78, 117, 21, 84, 69, 82, 74, 27, 28].

CNNs are computationally intensive to train, especially for the most challenging tasks. For example, state-of-the-art CNNs [35, 22, 20] perform billions of floating-point operations on each training input, and must process millions of inputs to attain good task accuracies. The time to train a model is therefore on the order of days or weeks. Moreover, because CNNs are designed iteratively, it may require multiple months to reach a final model after multiple training iterations.

Our work focuses on developing an automatic framework for improving CNN performance on multi-core CPUs. The motivation for this focus (as opposed to a focus on GPUs or FPGAs) is twofold: 1) Modern multi-core CPUs are easily accessible and they can have more than a teraflop of peak performance (Eg. Xeon Processor E7-8895 v3), which is enough to train medium and moderately large CNNs in reasonable time.
There is an abundance of multi-core CPU clusters — which are used in state-of-the-art large CNNs [35, 20] — that are easily accessible to both academic community and industry. Performance improvements on CPUs therefore simultaneously benefits many low-, mid-, and high-end users.

### 5.1.1 Convolutional Neural Networks

In a traditional ANN, the input for a recognition problem is represented as an array of floating-point values (e.g., pixels for image classification) and passed into a stack of layers. Each layer computes a non-linear function of either the problem’s input or the output of the previous layer. The multi-dimensional output of a layer is composed of neurons, where each neuron first computes a linear function (parameterized by a set of coefficients or weights) of all the outputs of the previous layer. The neuron then applies a non-linear function to that result.

A CNN differs from a traditional ANN in that each neuron connects to only a small region of the previous layer. Moreover, the neurons of a layer in a CNN are grouped into feature maps, in which all neurons in a feature map apply the exact same function (or feature) to its corresponding input region.

For CNN training on multi-core CPUs, state-of-the-art CNN infrastructures such as CAFFE [70], Theano [12], Torch7 [26], Chainer [?], CNTK [?], and TensorFlow [3] all implement convolution computation using a process we refer to as Unfold+Parallel-GEMM [18], where inputs are unfolded to cast it as a matrix-multiply (MM). The resulting MM is computed efficiently by linking to third party Basic Linear Algebra Subprogram Libraries (BLAS) such as MKL [64], ATLAS [139] or OpenBLAS [143],
Figure 5.1: CNN performance characteristics of using Unfold+Parallel-GEMM as a function of AIT (≈ 2 × Number of features) and sparsity with respect to scalability, goodput and single core performance. The design space is divided into six regions. The figure shows their different performance characteristics and the optimization techniques of our work (depicted in bold) for improving them. In addition, it illustrates regions that real world image classification benchmarks (ImageNet-22k, ImageNet-1K, CIFAR-10, MNIST) occupy in the convolution space.

which offer highly optimized Parallel-GEMM (General Matrix Multiply) implementations for multi-core CPUs.
5.1.2 Performance Characterization

Fig. 5.1 presents a characterization of the performance of Unfold+Parallel-GEMM as a function of the arithmetic intensity and sparsity of the CNN’s computation.

- Arithmetic Intensity (AIT) — the ratio of number of arithmetic operations to the number of memory operations in a computation. A high AIT is necessary to get high performance because memory operations are slower than arithmetic operations.

- Sparsity — the fraction of elements in a data array that are zeros. High sparsity in the data means that a naive execution approach performs many computationally intensive operations that could instead be elided without affecting the computation’s correctness.

Based upon the AIT and Sparsity of a convolution computation, we define three performance characteristics of Unfold+Parallel-GEM: scalability, single core performance, and goodput.

Scalability: Unfold+Parallel-GEMM scales poorly when the MM corresponding to a convolution computation does not have high AIT to begin with (Fig. 5.1, Region 2, 3, 4 and 5). We show that as we increase the number of cores, the number of arithmetic operations in Parallel-GEMM per core reduces proportionately, while the number of memory operation does not, resulting in a reduced AIT per core.

Single Core Performance: While large convolutions have adequate AIT to achieve high single core performance using Unfold+Parallel-GEMM (Fig. 5.1, Region 0, 1, 2 and 3), small and medium convolutions exhibit poor single core performance due to very low AIT (Fig. 5.1, Region 4 and 5). We show that the unfolding increases the number of memory operations, and thus reduces AIT.
Goodput: We define *goodput* as the rate of useful work in a computation. For example, it is possible to simultaneously achieve high throughput and low goodput if most of the computation is avoidable. Because CNN computations often involve sparse data, Unfold+Parallel-GEMM’s dense MM results in poor goodput. (Fig. 5.1, Region 1, 3 and 5).

### 5.1.3 Optimization Framework for CNNs

We present *spg*-CNN, an optimization framework for CNNs that achieves high Scalability, Performance and Goodput. *spg*-CNN consists of three key components that work collectively to address the three limitations of Unfold+Parallel-GEMM.

- **GEMM-in-Parallel**: *spg*-CNN improves scalability over Parallel-GEMM by running multiple instances of single-threaded GEMM in parallel (Fig. 5.1 Region 2, and 3). This simple re-scheduling prevents AIT reduction when scaling to multiple cores.

- **Stencil-Kernel**: *spg*-CNN improves single-core performance over Unfold+Parallel+GEMM for small convolutions where unfolding results in very low AIT. We develop a new approach inspired by stencil computations [34], which computes CNN through direct convolution without unfolding, exploits spatial reuse of inputs and improves AIT (Fig. 5.1 Region 4 and 5).

- **Sparse-Kernel**: *spg*-CNN improves the goodput of sparse CNN computations over Unfold+Parallel-GEMM and its sparse variants [64]. Sparse GEMM libraries are effective when both input matrices are highly (> 95%) sparse [89], but the inputs of CNN computations are typically a moderately (50 – 95%) sparse matrix and a dense one. *spg*-CNN incorporates a *pointer shifting* technique to compose a sparse
convolution as a series of small and dense MMs, performing computation in place without unfolding. Additionally, it generates efficient SIMD instructions, and exploits locality enhancing sparse data structures, to achieve high goodput on sparse and dense inputs.

Given a CNN, \textit{spg-CNN} generates codes and chooses the fastest among Parallel-GEMM, GEMM-in-Parallel, Sparse-Kernel and Stencil-Kernel for the forward-propagation (FP) and back-propagation (BP) phases of each layer, optimizing the training time. For CNNs with different network structures, \textit{spg-CNN} generates the best configurations based on their performance characteristics.

### 5.1.4 Contributions

This paper makes the following contributions:

- **Performance Characterization.** We systematically analyze the CNN performance of using Unfold+Parallel-GEMM with respect to scalability, single-core performance and goodput in a clear design space based on AIT and sparsity, show its limitations and identify the root causes (Section 5.3).

- **Optimization Framework.** We develop \textit{spg-CNN}, an optimization framework for CNNs, which introduces and integrates three key components, i) an alternate schedule using GEMM-in-Parallel, improving scalability, ii) a stencil-based code generator, improving per-core performance, and iii) a sparse code generator, exploiting sparsity and improving goodput, into a unified code generator that produces optimized codes for various CNN computations (Section 5.4).
- Evaluation: We evaluate spg-CNN using the CNN training frameworks Adam [20] and Caffe [70]. The results show performance improvements of up to 16x on real-world convolutions and an end-to-end speed up of 8.3x on CIFAR-10 image recognition benchmark (Section 5.5).

5.2 Background

In this section we present the basic elements of Artificial Neural Networks, Convolution Neural Networks, and the contemporary methods for training these networks.

5.2.1 Artificial Neural Networks (ANN)

An ANN comprises of a stack of layers each of which is a group of neurons.

**Layers and Neurons:** A layer, denoted by $A_{l+1}$, is a group of neurons in which each neuron in the layer computes a non-linear function of the outputs of neurons in the preceding (lower) layer. A neuron $A_{l+1}[i]$ specifically applies a non-linear function ($\phi$) to a linear combination of the outputs of the layer below ($A_l$) using a set of weights ($W$).

$$A_{l+1}[i] = \phi \left( \sum_j A_l[j] \times W[i, j] \right)$$

(5.1)

**Forward Propagation (FP):** FP evaluates an entire network to compute the network’s result on an input. It computes the network’s result by successively computing the output activations of the neurons of each layer based on the previous layer activations.

**Backward Propagation (BP):** BP computes the error gradients of a network’s weights with respect to a loss function. It works backwards from the output layer,
Figure 5.2: (a) An example of a 2D convolution using a $2 \times 2$ kernel with 2 input features of size $3 \times 3$ and 2 output features. (b) Unfolding a 2D image with two channels for convolving with a $2 \times 2$ convolution kernel. (c) $MM = W \cdot U^T$ representing a 2D convolution using a $2 \times 2$ kernel with 2 output features.
using the chain rule to compute the error gradient of each weight in a lower layer as a function of the upper layers.

**Stochastic Gradient Descent:** ANNs are often trained using *stochastic gradient descent* (SGD). For an input example and corresponding target output, stochastic gradient descent executes FP to compute the network’s output and then executes BP to compute error gradients of the weights. SGD then multiplies the error gradient of each weight by the value of the input connected to that weight to produce the *delta weights*. The delta weights are the updates that SGD applies to the existing weights to produce an updated model. Training then proceeds by repeating the procedure on a new input example with the updated model.

### 5.2.2 Convolutional Neural Network (CNN)

A CNN is a sub-class of ANNs where neurons in a layer are only connected to neurons in its local surroundings in the previous layer, and the weights are shared.

**Layers and Neurons:** Fig. 5.2a shows an example of a convolution on a two dimensional image of size $3 \times 3$ with two input features (Channel 0 and Channel 1), corresponding to, for example, the red and blue channels of the image. The convolution has two output features, Feature 0 and Feature 1. Each feature has individual sets of weights that correspond to each input feature. The weights for the first feature are the top two matrices under the Weights column whereas the weights for the second feature are the bottom two matrices. The kernel size of each output feature is $2 \times 2$, which is the size of each weight matrix.

To produce the first element of Feature 0, the convolution computes the inner product of the sub-region of Channel 0 within the black boundary and the feature’s
weights that correspond to that channel. The convolution then sums this result with
the inner product of the sub-region of Channel 1 within the black boundary and the
feature’s weights that correspond to that channel.

**FP:** A fully general convolution operation in two dimensions is represented using
a convolution kernel of 5-tuple \(<N_f, F_y, F_x, s_y, s_x>\). This convolution operation can be
mathematically written as

\[
O[f, y, x] = \sum_{c, k_y, k_x=0}^{N_c, F_y, F_x} I[c, y \times s_y + k_y, x \times s_x + k_x] \times W[f, c, k_y, k_x]
\]  

(5.2)

where \(O\) and \(I\) represent the output and input activations, \(y\) and \(x\) are the spatial
coordinates of the output activation, \(f\) represents the features of the output activations,
\(c\) represents the features of the input activations, \(s_y\) and \(s_x\) are the strides along
\(y\) and \(x\), and \(k_y\) and \(k_x\) represents the kernel coordinates (weights corresponding to
connections that are a distance of \(k_y\) and \(k_x\) from the output neuron along \(y\) and \(x\)
dimensions). We use \(N_f, N_c, F_y\) and \(F_x\) to represent the number of output features,
number of input features, kernel width along \(y\) dimension and kernel width along \(x\)
dimension, respectively.

**BP and SGD:** Just as with ANNs, SGD updates the network weights using
the error gradients computed by BP. For a given layer, Eq. 5.3 computes the error
in the input activations \((E_I)\) based on the errors of the output activations \((E_O)\), and
Eq. 5.4 computes the corrections to the weights \((dW)\) using the input activations \(I\)
and the error in the output activations \((E_O)\). \(N_y\) and \(N_x\) represent the spatial size of
the output activations along \(y\) and \(x\) dimensions.
\[ E_I[c, y, x] = \sum_{f, k_y, k_x=0}^{N_f, F_y, F_x} E_O[f, \frac{y - k_y}{s_y}, \frac{x - k_x}{s_x}] \times W[f, c, k_y, k_x] \]  \hspace{1cm} (5.3)

\[ dW[f, c, k_y, k_x] = \sum_{y, x=0}^{N_y, N_x} E_O[f, y, x] \times I[c, y \times s_y + k_y, x \times s_x + k_x] . \]  \hspace{1cm} (5.4)

### 5.2.3 Executing CNNs

The state-of-the-art execution method for CNNs is to use a two-step process \cite{18} that we term Unfold+Parallel-GEMM. We describe this method in FP as below, and BP calculations are done with similar transformation but in the reverse order.

1. **Unfold:** The first step is to unfold the input activation vector into a matrix. Fig. 5.2b illustrates the input activation unfolding procedure for the convolution presented in Fig. 5.2a. For each input channel, the unfolding procedure flattens the inputs to each kernel application into a row vector. The sequential concatenation of each row vector produces the unfolded representation of a channel. In the last step of unfolding, the procedure stacks each unfolded input channel from left to right to produce the final unfolded input matrix.

2. **Matrix Multiply:** The second step performs a matrix-matrix multiplication, with one matrix consisting of the layer’s weights and the other consisting of the unfolded activations, as shown in Fig. 5.2c. It constructs the weight matrix by stacking row vectors that correspond to the flattened representation of the weights for each feature. For example, the black box of the matrix titled “Weights" in the figure highlights the flattened representation of the weights of the first feature (which includes weights for both Channel 0 and Channel 1).
5.3 Performance Characterization

We categorize CNNs based on arithmetic intensity (AIT) and sparsity, and characterize the performance of using Unfold+Parallel-GEMM in terms of 1) single-core performance, 2) multi-core scalability and 3) goodput. This study exposes a range of performance issues of the current approach and guides the design of spg-CNN.

We produced all of the experimental results presented in this section on an Intel(R) Xeon(R) CPU E5-2650 with 16 cores with a peak performance of 41.6GFlops per core. For MM, we used the OpenBLAS library.

5.3.1 Single-Core Performance

Unfold+Parallel-GEMM’s unfolding procedure can reduce the maximum achievable fraction of the intrinsic AIT of the convolution operation, therefore resulting in poor single core performance (Fig. 5.1 Region 4 and 5).

The AIT of a convolution is \( \frac{|A|}{|I| + |W| + |O|} \), where \(|A|\) is the number of arithmetic operations and \(|I| + |W| + |O|\) is the number of memory accesses. The sets \( I, W, \) and \( O \) correspond to the input activations, weights, and output activations, respectively. Their sizes are calculated as follows:

\[
|A| = 2N_fN_xN_yN_cF_yF_x \quad (5.5)
\]

\[
|I| = N_xN_yN_c \quad (5.6)
\]

\[
|W| = N_fF_xF_yN_c \quad (5.7)
\]

\[
|O| = N_f(N_x - F_x + 1)(N_y - F_y + 1) \quad . \quad (5.8)
\]

The unfolding procedure increases the size of the activations by approximately a factor of \( F_xF_y \). In addition, the unfolded inputs need to be stored before the
MM, doubling the number of memory access to the unfolded input. Therefore, the minimum number of memory accesses of Unfold Parallel-GEMM is \(2|U| + |W| + |O|\), where \(U\) is the unfolded inputs with size

\[
|U| = (N_x - F_x + 1)(N_y - F_y + 1)N_c F_x F_y.
\]

The resulting fraction of the intrinsic AIT of convolution that Unfold Parallel-GEMM can achieve is at most \(r\), where

\[
r = \frac{|I| + |W| + |O|}{2|U| + |W| + |O|}.
\]

There are two dominant axes on which the fraction of intrinsic AIT achieved via Unfold Parallel-GEMM changes:

- **Kernel Size:** When convolution kernel size is much smaller than the input dimensions (with \(F_x \ll N_x\) and \(F_y \ll N_y\)), increase in the kernel size reduces \(r\). However, as the size of the convolution kernels approaches the size of the input dimensions (with \(F_x = N_x\) and \(F_y = N_y\) at the limit), the convolution itself more closely resembles a matrix multiply \((r \approx 1)\).

- **Output Feature Count:** As output feature count \(N_f\) increases, the number of memory accesses in the convolution are dominated by those to the weights \((r \approx 1)\).

Table. 5.1 illustrates the relationship between these dimensions and the AIT of Unfold+Parallel-GEMM. Specifically, Unfold+Parallel-GEMM achieves a higher fraction of the intrinsic AIT of the convolution for larger kernel sizes and larger output feature counts because either little unfolding is required (larger kernel size) or the overhead of unfolding is diminished by the size of the weights (output feature count).
Table 5.1: Different Convolutions, their intrinsic AIT, the AIT corresponding to Unfold+GEMM, and the region in Fig. 5.1 that they belong to. These benchmarks were chosen to represent convolutions with high, moderate and low AIT. For varying levels of sparsity, these six benchmarks cover the entire performance characterization space shown in Fig. 5.1.

<table>
<thead>
<tr>
<th>ID</th>
<th>(N_x(=N_y), N_f, N_c, F_x(=F_y))</th>
<th>Intrinsic AIT</th>
<th>Unfold+GEMM</th>
<th>Region (Reg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32,32,32,4</td>
<td>362</td>
<td>25</td>
<td>4,5</td>
</tr>
<tr>
<td>1</td>
<td>64,1024,512,2</td>
<td>2015</td>
<td>725</td>
<td>0,1</td>
</tr>
<tr>
<td>2</td>
<td>256,256,128,3</td>
<td>1510</td>
<td>226</td>
<td>2,3</td>
</tr>
<tr>
<td>3</td>
<td>128,128,64,7</td>
<td>3561</td>
<td>113</td>
<td>2,3</td>
</tr>
<tr>
<td>4</td>
<td>128,512,256,5</td>
<td>6567</td>
<td>456</td>
<td>2,3</td>
</tr>
<tr>
<td>5</td>
<td>64,64,16,11</td>
<td>1921</td>
<td>44</td>
<td>4,5</td>
</tr>
</tbody>
</table>

5.3.2 Multicore Scalability

On shared-memory multicore systems, large MM achieves near peak performance, and scales perfectly to all cores using Parallel-GEMM (Fig. 5.1 Region 0 and 1). However, there is a subspace of MM that renders sub-optimal performance, or scaling, or both (Fig. 5.1 Region 2, 3, 4 and 5), because of low AIT per core.

Fig. 5.3a presents the absolute performance per core and scalability for Parallel-GEMM corresponding to different convolution operations. We timed the execution of three MM corresponding to FP, gradient calculations and delta-weight calculations and calculated the GFlops per core. Each curve in the plot shows a different convolution.

Fig. 5.3a shows that Parallel-GEMM does not scale linearly in the number of cores. The AIT of MM depends on the size of the matrix. Specifically, MM of two square matrices of size \(n \times n\) has \(2n^3\) arithmetic operations and at least \(3n^2\) load + store operations. MM therefore has an AIT of at most \(\frac{2n}{3}\). While scaling Parallel-GEMM to multiple cores evenly divides the total number of operations of the MM across the
cores, it does not do the same for loads to and stores from per core private memory. This reduces the per core AIT.

**AIT per Core**: This is calculated by dividing the total amount of computation done per core by the number of memory load and stores to its private memory (L2 cache).

For instance consider a square MM where two square input matrices $A$ and $B$ of size $n \times n$ are multiplied together to produce matrix $C$ also of size $n \times n$. The AIT of this MM is $\frac{2n^3}{3}$. On a dual-core machine, each core can compute half of output matrix $C$. Computing half of $C$ requires either half of $A$ and entire $B$, or entire $A$ and half of $B$, depending on whether $C$ is partitioned along row, or column. Therefore, the AIT per core is $\frac{0.5 \times 2n \times n \times n + n \times n + 0.5n \times n}{0.5n \times n + n \times n + 0.5n \times n} = \frac{n^2}{2}$ resulting in a reduction from $\frac{2n^3}{3}$. Scaling to larger number of cores further reduces the AIT per core. Thus, we see poor scalability in Fig. 5.3a as we increase the number of threads.

---

**Figure 5.3**: (a) Scalability of Parallel-GEMM on up to 16 cores for convolutions shown in Table 5.1. (b) Sparsity across multiple epochs for three CNN benchmarks.
5.3.3 Goodput

The conventional approach to BP uses dense MM to compute error activations and delta-weight updates. One of the inputs to both of these calculations is the output activation error, denoted as \( E_O \) in Eq. 5.3 and Eq. 5.4. \( E_O \) represents corrections to the original activations of the layer. In many neural network models, these corrections tend to be sparse, with few non-zero elements. When a network exhibits high sparsity, dense MM is inefficient because a significant portion of the computation is multiplying and adding zero values.

In the presence of sparsity, we define efficiency in terms of goodput. Goodput is the rate of doing non-zero computation (Eq. 5.9). By assuming that only the output activation error in BP calculation is sparse, we establish an upper limit on the goodput of Parallel-GEMM (Eq. 5.10), which we refer to as the goodput of Parallel-GEMM based BP.

\[
\text{Goodput} = \frac{\text{Number of NonZero Flops}}{\text{TimeElapsed}} \quad (5.9)
\]

\[
\leq (1.0 - \text{Sparsity}) \times \text{Throughput} . \quad (5.10)
\]

For BP of real world DNN benchmarks, the goodput of Parallel-GEMM is significantly smaller than throughput because there is high level of sparsity in BP. Fig.5.3b shows sparsity in activation errors across multiple training epochs for three real world benchmarks, MNIST, CIFAR and ImageNet-100. After the second epoch, all three benchmarks have a sparsity level of more than 85% in their activation errors. Further, as the model becomes more accurate, these activation errors become even sparser.

At this level of sparsity, dense GEMM goodput is at most 15% of the throughput. For example, if dense GEMM has a throughput of 60 GFops per core, then the goodput
is only 9 GFlops, while the remaining 51 Gflops is wasted in zero calculations. In Fig. 5.1, regions 1, 3 and 5 have poor goodput.

**Figure 5.4:** The set of figures shows absolute performance for dense convolutions, goodput for sparse convolutions, and relative speedups over baselines using each of the three techniques of *spg*-CNN. We present the results of 6 different convolutions in Table. 5.1. The results indicate that our techniques achieve significant performance improvement in Regions 2 (Fig. a, b) and 4 (Fig. c, d), and significant goodput improvement in Regions 1, 3, and 5 (Fig. e, f).
5.4 Optimization Framework

Sec. 5.3 motivates $spg$-CNN, which introduces and integrates the following three techniques into a unified optimization framework, generating optimized codes covering various CNN computations with different characteristics.

**Stencil-Kernel:** Improve single-core performance for convolutions with smaller kernel sizes and fewer output features through custom-generated stencil-based kernels for FP. These kernels avoid unfolding and therefore avoid the AIT reduction that unfolding introduces.

**GEMM-in-Parallel:** Improve multicore scalability by running single-threaded GEMM on multiple training inputs in parallel. This technique avoids the AIT reduction that arises from partitioning the computation via Parallel-GEMM.

**Sparse-Kernel:** Improve the goodput for sparse models through custom-generated sparse-based kernels for BP. These kernels elide computations on zero values that have no effect on the convolution’s result.

We next describe each of our techniques in the order of their relative complexity to facilitate readability.

5.4.1 GEMM-in-Parallel

We improve the scalability of convolutions on multi-core hardware using GEMM-in-Parallel: multiple instances of single-threaded GEMM run concurrently on different cores. GEMM-in-Parallel performs better than Parallel-GEMM because inputs are not divided across cores, and so, per-core AIT and (therefore performance) stay the same.
calability of convolution can be improved by using GEMM-in-Parallel, where many instances of single-threaded GEMM run concurrently on different cores instead of using multi-threaded GEMM in sequence. For instance, if we want to perform convolution on \( n \) images, it is more scalable to use single-threaded GEMM, and use data parallel libraries such as OpenMP to parallelize across multiple images than it is to run Parallel-GEMM and run each image in sequence. The reason is that parallelizing across multiple images does not decrease AIT per core. On the other hand, as we explained in Sec. 5.3.2 parallelizing the convolution itself across multiple cores reduces AIT per core as we increase the number of cores.

We can use this technique to improve performance in two ways. i) Mini-Batch Training: In mini-batch training the input data is processed in batches instead of Mini-batch means training a large sample of input data in sets of small batches of inputs instead on training one input at a time. During training with mini-batching, output activations are computed for the entire mini batch before computing the error or the delta weights.

To improve scalability, we can use OpenMP to parallelize convolution of multiple inputs within a mini-batch and convolve each input using sequential GEMM. While doing this, we can make the mini-batch size a multiple of the number of compute cores to ensure load-balancing. This technique guarantees better performance without affecting accuracy as this is only a change of computation schedule and not the actual computation itself.

ii) Data-Parallelism: This is an alternate way of parallelizing DNN computation. In data-parallelism, each thread works in parallel with other threads on a unique input and shares weights across all threads. This method of training automatically
utilizes parallelization of sequential GEMM as each thread is working in parallel with all other threads on different images.

**Evaluation:** Fig. 5.4a shows the scalability of GEMM-in-Parallel, depicting the absolute performance per core with varying core number from 1 to 16. The results show the performance per core is roughly steady, and drops by $< 15\%$ on average. In contrast, the average performance drop per core for Parallel-GEMM is $> 50\%$ (Fig. 5.3a). Fig. 5.4b shows the relative speedup of GEMM-in-Parallel over Parallel-GEMM. The relative speedup grows with more cores, indicating better scalability. Also, convolutions with fewer output features benefit more from GEMM-in-Parallel because their low AIT is further reduced by Parallel-GEMM on more cores.

5.4.2 Sparse-Kernel

Our framework takes advantage of the sparsity in error gradients to improve BP goodput by using sparse kernels to compute error gradients and weight deltas. These kernels exploit efficient data representation, vectorization, cache and TLB optimization as well as a new pointer shifting technique, to achieve high goodput on sparse and dense inputs. Sparse convolution is performed in place, without unfolding, as a composition of small and dense MMs.

**Sparse Data Representation:** Error gradients are stored in Column Tiled-Compressed Sparse Row (CT-CSR), our adaption of the popular CSR format for sparse matrices. In CSR, a sparse matrix is stored using three arrays: i) value array for storing non-zero values, ii) column index array for storing column indices of the non-zero values, and iii) row index array for storing the starting position of each row in the data array or column index array. In CT-CSR, the sparse matrix is tiled along
columns, and each tile is stored in CSR (see Fig. 5.5a). CT-CSR provides better locality than CSR by tiling along both row and column of the sparse matrix to enable greater reuse of tile elements (in both dimensions).

Another advantage of CT-CSR is the reduction in TLB misses when accessing elements within a tile. In CT-CSR elements of two adjacent rows within a tile are also adjacent in memory. Without this explicit tiling, elements corresponding to two adjacent rows may be far apart depending on the column width of the entire matrix requiring two TLB lines to access them. Thus, CT-CSR reduces the number of TLB entries required to hold the tile in cache, resulting in a reduction in TLB misses.

*spg*-CNN generates efficient AVX vector instructions using Intel Intrinsics, that vectorize across the dense input, optimize for cache locality, and reduce TLB misses. Our locality optimization techniques are similar to [53].
Vectorization: Our code generator uses sparse-dense matrix multiplication as a basic code block for the generated code to efficiently execute the convolution with vectorization and without (un)folding. The output of these basic code blocks are computed in place without unfolding using a novel technique that we call pointer shifting.

For illustration, consider the error gradient calculation in Eq. 5.3. First we identify the MM operations within this calculation. We rewrite the equation as Eq. 5.11, where $S[c, y, x, k_y, k_x]$ is given by Eq. 5.12. For a fixed value of $k_y$, $k_x$, $y$ and $x$, Eq. 5.12 reduces to Eq. 5.13, which is a matrix-matrix multiply.
Because $E'_O$ is sparse, and $W'$ is dense, Eq. 5.13 can be computed efficiently by vectorizing along channels ($c$), as shown in Fig. 5.5b. We first perform data layout transformation: the weights ($W'$) and outputs ($E_I$ or $S'$) are transformed so that $c$ is the fastest varying dimension in memory, and input ($E_O$ or $E'_O$) is transformed so that $f$ is the fastest varying dimension in memory. Then each non-zero element $E'_O[f]$ is multiplied with the corresponding vector $W'[f,*]$, shown using the bold black boxes to produce a vector of $S'[*]$, where $*$ represents $c = 0,1,2$ stored as a vector.

Now, consider all the inputs $E_O[y',x',f]$ contributing to output vector $E_I[y,x,*]$. This can be written as

$$E_I[y,x,*] \leftarrow E_O[f, \frac{y-k_y}{s_y}, \frac{x-k_x}{s_x}]$$

where $y' = \frac{y-k_y}{s_y}$ and $x' = \frac{x-k_x}{s_x}$ for a given value of $k_y$ and $k_x$. In other words, each input value $E_O$ contributes to multiple output vectors $E_I$, given by

$$E_O[y',x',f] \rightarrow E_I[y's_y+k_y,x's_x+k_x,*] .$$

Using this relation, we can identify the position of the output vector $E_I[y,x,*]$ for a given input $E_O[f,y',x']$, and kernel coordinates $k_y$ and $k_x$. We can thus compute
sparse MM given by Eq. 5.13 in place, for all values of $k_y$ and $k_x$, without unrolling them. This is shown in Fig. 5.6. Each arrow between $E_O$ and $W$ represents a sparse MM between input $E[y', x', f]$, weights $W[k_y, k_x, f, *]$, for different values of $k_y$ and $k_x$. The arrows between $W$ and $E_I$ shows the position of the output vector resulting from the sparse MM.

Evaluation: We refer to the code generated by our sparse code generator as Sparse-Kernel (BP). Fig. 5.4e shows its goodput for various levels of sparsity on 16 cores. The costs of data-layout transformations and creating the CT-CSR representation are included. For $< 90\%$ sparsity, our kernels achieve consistently high goodput for all sizes of the convolutions. The performance drop after $> 90\%$ sparsity is due to the bottleneck shifting from error gradient computation to data-layout transformations, but the goodput is still much higher than Unfold+Parallel-GEMM as we see next.

Fig. 5.4f shows the relative speedup of our sparse kernel over Unfold+Parallel-GEMM. With sparsity $>= 0.75$, we consistently outperform. With sparsity of $>= 0.90\%$, we are significantly faster with $3x - 32x$ speedup. The much larger improvement for blue and green kernels (first and last) is not just due to sparsity but also because Sparse-Kernel (BP) does not reduce AIT of the convolution by casting it into MM as described in Sec. 5.3.1.

5.4.3 Stencil-Kernel

We develop a new approach inspired by stencil computations [34], which computes CNN through direct convolution without unfolding, exploits spatial reuse of inputs and improves AIT. In stencil computation [34], each element of an array is updated
based on the neighbouring values specified by the stencil. For example, consider the following three-point stencil in one dimension:


(5.16)

Each element of \( A \) is used to compute three output elements (e.g., \( A[x+2] \) is used to compute \( A[x] \), \( A[x+1] \) and \( A[x+2] \)). This \textit{spatial reuse} property of stencils improves AIT by reusing an input value, while it is in fast memory, to compute multiple output values. Convolutions also exhibit spatial reuse since each input neuron contributes to multiple neighboring output neurons. Thus, stencil computation is more efficient for convolutions than unfolding which destroys spatial result by replicating the input neurons. Although there is a rich body of work exists on optimizing stencil computations [101, 58, 76, 60, 104], we present the first work to extend those optimization techniques for training CNNs on CPUs.

We illustrate how to use stencil computation for FP with the following example:

\[
O[f,y,x] = \sum_{c,ky,kx} I[c, y + ky, x + kx] \times W[f, c, ky, kx]
\]

(5.17)

\[
= \sum_c \left( \sum_{ky,kx} I[c, y + ky, x + kx] \times W[f, c, ky, kx] \right)
\]

(5.18)

\[
= \sum_c (S[f, c, y, x])
\]

(5.19)

where, \( O, I, W \) are output activations, input activations and weights, respectively.

For a given \( y, x, c \) and \( f \), the computation inside the parenthesis in Eqn. 5.18 is a two dimensional \( F_x \times F_y \) point box stencil operation. \( S[f, c, y, x] \) represents the result of this stencil operation. For a given \( f, c \) we can simplify and write the stencil operation
as

\[ S[y, x] = \sum_{k_y, k_x} I[y + k_y, x + k_x] \times W[k_y, k_x] \]  

(5.20)

Our FP kernel generator uses this stencil operation as the building block for generating efficient vector code. It consists of two components: i) a basic block generator, ii) a schedule generator. The basic block generator generates register tiled vector instructions to improve the reuse of each input vector load and reduce the total number of load instructions. The schedule generator tiles the generated computation blocks to optimize for cache locality and TLB misses.

**Listing 5.1:** Basic code block for \(1 \times 2\) stencil with a register tile size of \(r_x=1\) and \(r_y=2\)

```c
/*load input vector 0 and compute 1 contribution*/
m256 ivec0 = mm256_loadu(input + (y + 0)*NX + x);
wvec[0][0] = mm256_set1(weight[0]);
ovec[0][0] = mm256_add(ovec[0][0], tmpvec);

/*load input vector 1 and compute 2 contributions*/
m256 ivec1 = mm256_loadu(input + (y + 1)*NX + x);
wvec[0][1] = mm256_set1(weight[1]);
temp0 = mm256_mul(ivec1, wvec[0][1]);
ovec[0][0] = mm256_add(ovec[0][0], tmpvec);
ovec[0][1] = mm256_add(ovec[0][1], temp0);
/*load input vector 2 and compute 1 contribution*/
m256 ivec2 = mm256_loadu(input + (y + 2)*NX + x);
temp1 = mm256_mul(ivec2, wvec[0][1]);
ovec[0][1] = mm256_add(ovec[0][1], temp1);
```

Basic Block Generator: The basic block generator vectorizes the stencil computation along \(x\) dimension and generates the instructions within a block as follows. For an output vector register tile of width \(r_x\) and height \(r_y\), the block generator identifies all the input vectors that contribute to the tile. For each input vector, it generates instructions for loading it, and for computing its contributions to all the output vectors in the register tile. Lst. 5.1 shows the basic computation block for a convolution with \(F_x = 1\) and \(F_y = 2\), and a register tile size of \(r_x = 1\) and \(r_y = 2\). The block consists of three vector loads. The load of vector \(ivec0\) only contributes to one output
vector $ovec[0][0]$ in the register tile, while the load of $ivec1$ contributes to two vectors $ovec[0][0]$ and $ovec[0][1]$ in the output register tile. In other words, the load of $ivec1$ is reused twice. The shape and size of the register tile can change the overall reuse of each input vector load. In general, the size of $r_x$ and $r_y$ should be chosen such that $r_x r_y \leq$ number of physical vector registers and the number of load instructions is minimized. While this is a geometric optimization problem, our code generator finds optimal solution by iterating over all possible values for $r_x$ and $r_y$ meeting the first criteria as commodity machines typically have a relatively small number of vector registers.

**Strided Convolutions:** The code generation technique above works well for convolutions with unit stride along $x$ dimension; the convolutions with non-unit stride are more challenging because strided access can hinder effective vectorization. More specifically, for efficient vectorization, the inputs corresponding to an output vector should be contiguous in memory so that a single vector load instruction can load the inputs from memory to a vector register. However, any non-unit stride along $x$ dimension will require loading inputs into a vector that is not contiguous in memory. Thus, our code generator performs a data-layout transformation to get the required input contiguous in memory for effective vectorization. For a given stride $s_x$, the layout of the input is transformed as

$$I[f, y, x] \rightarrow I[f, y, s', x']$$

such that $s = x \mod s_x$, $x' = x/s_x$ and $N_x s + x' = x$, where $N_x$ is the size of the $x$ dimension. This data-layout transformation, inspired by [58], converts unaligned vector loads into aligned vector loads.
Schedule Generator: Locality optimizations are used to reduce TLB and cache misses. Corresponding input and output are copied into contiguous memory to reduce the required number of TLB entries for accessing them, and then tiled so that input and output tiles fit in cache.

**Evaluation:** Fig. 5.4c shows scalability and absolute performance of Stencil-Kernel(FP) (including the data-layout transformation time). We see that Stencil-Kernel (FP) scales better than GEMM-in-Parallel (Figure 5.3a) as the impact of increasing core count on the performance per core is small.

Fig. 5.4d compares the performance of Stencil-Kernel(FP) and GEMM-in-Parallel. Stencil-Kernel(FP) outperforms GEMM-in-Parallel for small convolutions (< 128 output features) because it improves their AIT. However, GEMM-in-Parallel performs better for larger convolutions, which already have large AIT. We present additional results for convolutions with non-unit strides in the Appendix (Fig. ??)

### 5.4.4 Putting it all together

*spg*-CNN integrates the three techniques and automatically identifies the best set for each convolution layer of CNNs with different characteristics. It runs each layer with Parallel-GEMM, GEMM-in-Parallel, and Stencil-Kernel(FP) for FP and Parallel-GEMM, GEMM-in-Parallel and Sparse-Kernel(BP) for BP. Based on the measured performance, it chooses the fastest technique to deploy for each layer. For BP, it checks for a change in relative performance between these techniques after a pre-specified number of epochs as error gradient sparsity changes during the training. This ensures performance optimality at all training phases.
For our implementation and machine, GEMM-in-Parallel scales better than Parallel-GEMM for layers with < 1024 features, Sparse-Kernel(BP) is faster than GEMM-in-Parallel for layers with > 75% sparsity, and Stencil-Kernel(FP) is faster than GEMM-in-Parallel for layers with < 128 output features. These numbers are sensitive to the parameters of the implementation and the machine.

5.5 Experimental Evaluation

We implement and evaluate our framework on state-of-the-art training platforms and show its performance improvement on the convolutional layers of four well-known benchmarks, as well as end-to-end performance of training CIFAR.

5.5.1 Methodology

Benchmarks: We use four popular image recognition CNNs in our experiments: MNIST (LeCunn) [83], CIFAR-10 [123], ImageNet-1K (AlexNet) [78], and ImageNet-22K (Adam-ImageNet) [20]. The convolutional layer specifications for these benchmarks, based on the corresponding publications, are summarized in Table ?? in Appendix. The MNIST model classifies 28x28 black-and-white hand-written digits into 10 categories, CIFAR-10 classifies 32x32 RGB images into 10 categories, while ImageNet-1K and ImageNet-22K classify 256x256 RGB images into 1000 and 22000 categories.

Training Platforms: We use the ADAM [20] and CAFFE [70] CNN training platforms to obtain baseline results of the conventional approach, which we label as Parallel-GEMM (ADAM) and Parallel-GEMM (CAFFE). The parallel-GEMM implementations in ADAM and CAFFE use Intel MKL [64] and OpenBLAS [143]
GEMM libraries respectively. Our results show that the limitation of the conventional approach is independent of the specific training platforms and GEMM library implementations.

Our Framework: We implement our framework on top of ADAM. To show overall performance as well as the incremental contributions of individual techniques, we present the results for i) GEMM-in-Parallel for both FP and BP ii) GEMM-in-Parallel for FP and Sparse-Kernel for BP, and ultimately, iii) Stencil-Kernel for FP and Sparse-Kernel for BP.

Hardware: All of the experiments were run on a Xeon Intel(R) Xeon(R) CPU E5-2650 with 16 physical cores (32 logical cores with hyper-threading enabled).

5.5.2 Improvements of Real World Convolutions

Fig. 5.7 shows performance improvement using our framework over Parallel-GEMM for convolution layers in the four real-world benchmarks. We achieve 2x - 15x speedup on FP. More specifically, for ImageNet 22K and 1K, the speed up is a result of using GEMM-in-Parallel. These convolutions, with the number of output features ranging from 96 to 384, do not have enough AIT for Parallel-GEMM to be effective, but enough for GEMM-in-Parallel to perform well. For CIFAR and MNIST, we achieve even higher speedup than ImageNet22K and 1K, by using Stencil-Kernel. Their number of features ranges between 20 and 64, resulting in low AIT. While using GEMM-in-Parallel improves AIT, it is still not highly effective. Stencil-Kernel increases AIT further and achieves higher performance. Take layer 1 of CIFAR as an example: speedup of GEMM-in-Parallel over Parallel-GEMM is 11.5x (shown as blue bar in Fig. 5.7), while Stencil-Kernel boosts the speedup further to 15x (green bar).
For smaller CNNs like MNIST, both Parallel-GEMM and GEMM-in-Parallel perform poorly while Stencil-Kernel achieves 9x speedup over them.

Last but not the least, Fig. 5.7 also shows performance improvement on BP using our Sparse-Kernel. These speedup values are based on 85% sparsity in errors. We pick this sparsity level conservatively based on Fig. 5.3b. We achieve speedup of 2 – 14x over the baselines.

### 5.5.3 End-to-End evaluation on CIFAR-10 training

Our framework accelerates the end-to-end training of CIFAR-10 by 12.3x and 8.3x compared to Parallel-GEMM(ADAM) and Parallel-GEMM(CAFFE) respectively. Figure 5.8 reports the performance (throughput) of the different techniques in terms of images trained per second. The x-axis shows the number of cores used for training.
For one and two cores, Parallel-GEMM (CAFFE) is the fastest, but for more than two cores, both Parallel-GEMM (CAFFE) and Parallel-GEMM (ADAM) stop scaling — due to decrease in AIT per core as discussed in Sec. 5.4.1. For more than 2 cores, GEMM-in-Parallel scales better than both Parallel-GEMM (CAFFE) and Parallel-GEMM (ADAM) — GEMM-in-Parallel does not reduce AIT as we increase the number of cores.

Fig. 5.8 also shows the performance improvement using Sparse-Kernel on BP and Stencil-Kernel on FP. At 32 cores, Sparse-Kernel on BP increases the number of images per second from 1,600 to 2,061, improving throughput of about 28%. Adding Stencil-Kernel to FP increases the throughput to 2,283, a further 10% improvement. Notice these are net improvements on the end-to-end performance of the system, while the improvement on convolution layers is even higher, up to 30%, as seen in Fig. 5.7.
Summary: Parallel-GEMM (CAFFE) and Parallel-GEMM (ADAMS) peak performance are 273 and 185 images per second, respectively. Using Stencil-Kernel (FP) and Sparse-Kernel (BP), we increase the training throughput to 2,283, a net speedup of 8.36x. To put this in perspective, it takes Parallel-GEMM (CAFFE) 36 mins to train our model, while the optimized version takes only 4.3 minutes.

5.6 Related Work

This section reviews the related work on optimizing CNNs from various perspectives.

Parallelism. There have been several efforts to parallelize and scale neural networks on a cluster of distributed machines with multicore CPUs [35, 20, 144]. In contrast, our work focuses on CNN optimization on a single multicore machine, and is, therefore, complementary to the above efforts.

On a multicore CPU, there are two forms of parallelism for CNN training, parallel-GEMM and GEMM-in-parallel. Many deep learning frameworks such as CAFFE, TensorFlow, Theano and Torch 7 use the first approach. Our study presents the first performance characterization of parallel-GEMM vs. GEMM-in-parallel for CNNs, and we show that while both approaches are similar in Region 0, GEMM-in-parallel is faster in Region 2, especially with increased number of cores. Alternatively, Caffe con Troll[4] improves Parallel-GEMM performance in Region 2 by batch partitioning, and executing each partition in parallel (one per core).

Sparsity. [135, 136] were the first to propose reducing computation in multilayer perceptrons by simply avoiding calculations with zero values, but they do not discuss how to implement this efficiently on modern architectures. The poster [49] describes
a sparse dense MM algorithm using compressed sparse row to store the sparse error/activation matrix. Their approach and evaluation is limited to large sparse MM that are not applicable to real CNNs. [88] also presents a sparse dense MM algorithm, which exploits sparsity in the weights. Their algorithm is based on knowing the position of non-zero elements in weights in advance to generate the sparse MM code, therefore, their approach is only applicable for CNN inference but not training. In contrast, our approach is designed for training and we exploit sparsity in activation errors. Additionally, instead of using sparse MM we compose a sparse convolution as a series of small and dense MMs, achieving high goodput on sparse dense inputs.

Direct convolution. Although Unfold+Parallel-GEMM is the most popular approach, CNN computation can also be implemented using direct convolution. For example, Intel Deep Learning Framework (IDLF) [63] implements direct convolution exploiting SIMD hardware on CPUs. Our work characterizes the tradeoff between the two approaches and analyzes the causes. We present a stencil-based direct convolution kernel for small CNNs while propose to use GEMM-in-Parallel for moderate/large CNNs. Although GEMM is a popular approach, CNNs can also be computed on CPUs via direct convolutions. For example, the Intel Deep Learning Framework (IDLF) [63] implements direct convolution exploiting SIMD hardware (AVX2 ISA) on CPUs. Our study of the tradeoffs of these two approaches reveals that direct convolution is better for small CNNs, and GEMM is better for moderate/large CNNs. Moreover, our direct convolution approach differs from IDLF in our use of stencil based optimizations. Unlike IDLF, our stencil kernel enhances vector register reuse.
and L1 cache locality by exploiting spatial reuse in convolutions. Other CNN frameworks such as cuDNN [19], NeuFlow [43], implement direct convolution on GPUs and custom hardware, but not on CPUs.

Other techniques. There are other complementary efforts to optimize CNN, e.g., by exploiting redundancies [36, 67, 37, 88], using FFT-based computation [91], and applying more efficient MM computation [29].

5.7 Conclusion

This chapter presents the first characterization of the optimization opportunities for training CNNs on CPUs. Given this characterization, we have designed and implemented an optimization framework spg-CNN that beats the state-of-the-art approaches to training CNNs by an order of magnitude. In an environment where competitive CNN models may take weeks to train, our results enable CNN model engineers to iterate an order-of-magnitude more quickly through the multiple model designs that are often required to identify a good model for one’s domain.
Chapter 6: On Fusing Recursive Traversals of K-d Trees

6.1 Introduction

Fusion is a key transformation for data locality enhancement. A common scenario where fusion can result in significant performance improvement involves a sequence of operations with a producer-consumer relationship—an earlier operation assigns to the elements of an intermediate data structure and one or more subsequent operations read these elements. The total number of elements in such intermediate data structures can be much larger than cache capacity, causing them to be evicted from cache to main memory after production and transferred back from main memory to cache when the consumer operations are executed. Instead of fully producing all elements of a data structure and then later executing the consumer operations, the statements in the producer and consumer operations could be combined into a single fused operation. With this fused structure, each produced data element can be immediately read by the fused consumer statement(s) before they are evicted to main memory.

Well-developed formalisms and algorithms exist for characterizing data dependences in loop computations, and sufficient conditions based on data dependences can be used to determine the validity of loop fusion [6, 47, 92, 33, 15, 103]. Thus, automatic code analysis and transformation via loop fusion for imperative programs
is implemented in production compilers. The elimination of (some) temporary trees in functional programs has been at the core of deforestation techniques [134, 72], but the fusion of operators on tree traversals expressed in imperative programs such as C/C++ poses numerous additional challenges, such as the characterization of dependencies and expressing the set of semantics-preserving fusions of DAGs of operations.

In this chapter, we develop a compile-time approach to dependence characterization and program transformation to enable fusion across recursively specified tree-traversal operators. Our work is motivated by the MADNESS (Multiresolution Adaptive Numerical Environment for Scientific Simulation) high-level software environment for the solution of integral and differential equations in many dimensions [90, 56]. A MADNESS user writes a program using high-level operators on functions over space (i.e., these functions are the “variables” of the MADNESS program). Examples of MADNESS operators on functions are addition, multiplication, convolution, and differentiation. Functions over space (the MADNESS program’s variables) are internally represented using k-d trees, refined to a tree depth based on the desired numerical precision for the computation. The implementation of MADNESS operators has a recursive specification over the structure of the k-d tree representation of the produced result variable, suitably traversing the k-d tree representations of the input operand variables. The k-d trees representing MADNESS variables are typically much larger than cache. Therefore, the execution of each MADNESS operator generally requires a considerable amount of data movement across nodes and within the memory/cache hierarchy of nodes on a distributed-memory cluster. However, it is not always necessary to completely form the entire k-d tree corresponding to an intermediate MADNESS variable before it is used as an input to other operators—under some conditions, the
production of an intermediate variable’s k-d tree can be interleaved with its use as an input to another operator. Such interleaved (fused) execution of the operators can result in significant reduction in the amount of data movement and synchronization.

Motivated by the opportunities and challenges in optimizing MADNESS programs, we formalize a broader class of recursive tree computations on k-ary trees for which we develop an approach for characterizing dependences, validity conditions for fusion across operators, and compile-time transformation for fused execution. While there has been work on dependence analysis and tiling for programs performing tree traversals [50, 81, 62, 109, 71, 137], we are unaware of any prior work that addresses compile-time transformations to fuse computations involving simultaneous traversal of multiple trees whose structures are only known at runtime.

The chapter makes the following contributions:

- It develops a data dependence characterization of a class of recursive computations traversing multiple k-ary trees.
- It develops necessary and sufficient conditions for fusion across a collection (directed acyclic graph) of k-ary tree operators with producer-consumer relationships.
- It develops a compiler algorithm for automatically generating code for fused composite recursive operators from an input program containing a DAG of primitive recursive operators.
- It demonstrates significant performance improvement over the production MADNESS environment using the new approach to create fused operators.
6.2 Background

6.2.1 \( k \)-d Trees

A function (value of fields) on a space of dimension \( k \), is encoded using a \( k \)-d tree that represents a specific partitioning of the space. For example, a binary partition tree can be used to represent a function in 1-d space.

Consider a function with a domain of \(-1\) to \(+1\). The root node of the tree represents this entire domain, while the two children node of the root represents each half of the domain, i.e. \(-1\) to \(0\) and \(0\) to \(+1\). In general, in a \( k \)-d tree, \( 2^k \) nodes at depth \( n \) with a common parent node at level \( n - 1 \) in the tree divides the space represented by the parent node into \( 2^k \) regions.

The amount of variation in the values of the function at a particular region in space determines the depth of the tree in that region. Each node stores coefficients that can be used to construct the function in the corresponding region. Fig. 6.1, shows an example of a 1D numerical function, and the spatial tree representing it.

**K-ary trees:** We consider \( k \)-ary trees, as a generalization of \( k \)-d trees, in which corresponding nodes in a tree represent related information (say the same spatial domain). Nodes A and B in two trees are said to correspond to each other if the they are both root nodes, or can be reached from the root nodes by the same path \( c_0, c_1, \ldots, c_n \), where \( c_i \) denotes the child of the node in the path at level \( i \) in the tree.

Given a node in a tree, we assume that an operation can access its corresponding node in other trees. From such a node, other nodes can be accessed through parent/child links or other unknown operations. In the rest of the chapter, we shall use the terms \( k \)-ary trees and spatial trees interchangeably.
6.2.2 Operations on k-ary Trees

As mentioned above, a (spatial) tree is represented as a collection of indexed and possibly labeled nodes. The actual implementation uses a map where $i[l]$ represents the node of label $l$ in tree $i$. With those notations, an operation on spatial trees from $i_1$ and $i_2$ to $o_1$, is analogous to an operation on a map, as illustrated by the two different operations below:

$$o_1[l].value = i_1[l].value + i_2[l].value \quad (6.1)$$

$$o_1[l].value = o_1[l.parent].value + i_1[l].value \quad (6.2)$$

The first example adds the values of $i_1$ and $i_2$ at node label $l$ and assigns it to $o_1$ at node label $l$. The second one adds the values of $i_1$ at node label $l$, and value of $o_1$
at the parent node of \( l \) and assigns it to \( o1 \) at label \( l \). Due to the hierarchical ordering between node labels in a spatial tree, it is convenient to represent the iteration space of a spatial tree operation recursively. For example, Lst. 6.1, shows a recursive specification for a tree operation given by Eq. 6.2.

A program written using spatial tree operations can have multiple input, output, and intermediate spatial trees. In order to relate the trees, we assume that the trees hierarchically partition the same space. Two trees are not necessarily isomorphic (as opposed to what was implicitly assumed in Lst. 6.1). However, each part of the space (represented by a node and its corresponding nodes in all trees) is associated with a unique label. Alternatively, each spatial trees involved in a program can be considered to constitute a unique field in a canonical spatial tree.

### 6.3 FuseT Compiler Framework

FuseT is a compiler framework for writing programs using tree operations. In FuseT, a tree operation is defined using a recursive specification. For a program written in FuseT, it performs a source-to-source code transformation to improve locality by fusing multiple tree operations in the program into a single recursive specification. In other words, it fuses multiple tree traversals into a single tree traversal.
6.3.1 Primitive Operators

A primitive operation is written in FuseT using a recursive template, given by

A **primitive operator** takes one or more spatial trees as input operands and produces a single spatial tree as output. It is a single-assignment operator. It is defined in terms of node operations, which describes the computation to be performed at each node label. The execution of an operator results in node operations being executed at some subset (possibly the full set) of canonical spatial tree nodes, producing data (which could be null) for the result spatial tree at that node.

A primitive operator given by Lst. 6.2 assigns values to a single field (*output*) in the canonical spatial tree. *vector<ST> pl* is the list of all the fields that the output field depends on. *computeOp* at label *l* can read the value of these fields at any label to compute the return value at label *l*. The *needRecursion* is a side effect free function that returns a control Boolean to identify if a recursive call
at the children node is necessary. It can read the value of any field in the pl or output. In our specification isPre is a Boolean that specifies when computeOp is executed. It allows for the recursive specification of an operator to have either a pre-order traversal, or a post-order traversal.

### 6.3.2 Data Access Types

The operator specification also specifies the *Data Access Type* for each of the field in the InputList and for the output field.

**Definition 6.3.1** (Data Access Type). *For a given operator, the data access type for an input or output field is the spatial relation from node label 1 to 1’ such that computeOp or needRecursion at label 1 reads the value of the field at label 1’ in the canonical spatial tree.*

The Data Access Type is defined in terms of *spatial relation (SR)* which specifies the relative position of one node label with respect to another. We define four mutually exclusive spatial relations (SR) between two node labels.

- **Same Node Relation (SNR):** 1’ = 1. In SNR, node label 1’ is same as node label 1 in the canonical spatial tree.

- **Ancestor Relation (AR):** 1’ = l.ancestor. In AR, node label 1’ is an ancestor node to node label 1 in the canonical spatial tree.

- **Descendent Relation (DR):** 1’ = l.descendent. In DR, node label 1’ is a descendent of a node label 1 in the canonical spatial tree.

- **Some Sibling Relation (SSR):** (l’ ≠ l) ∧ (l’ ≠ l.ancestor) ∧ (l’ ≠ l.descendent).

A pair of node label (l, l’) has SSR iff none of the spatial relation described
above (SNR, AR, or DR) exists between the two node labels in the canonical spatial tree.

Notice that any pair of label \((l, l')\) falls under exactly one of the four SR. As there are four spatial relations, there are also four data access types corresponding to each of these relation. These data access types for an operator and a field are used for identifying dependences between node computation when there are producer and consumer operations in a sequence of operations.

6.3.3 Examples

Lst. 6.3 and Lst. 6.4 show two operator specifications. Lst. 6.3 is an example of a recursive specification of an operation, which computes the value of the output field at label \(l\) as sum given by \(f_1[l] + \text{output}[l.\text{parent}]\). The data access to \(f_1\) for this operator is of the type SNR, as \(\text{output}[l]\) depends on \(f_1[l]\). Similarly, the data access to \(\text{output}\) for this operator is of the type AR, as \(\text{output}[l]\) also depends on \(\text{output}[l.\text{parent}]\).

This is an example where \(\text{isPre}\) is true. The AR data access to output field causes a flow dependence, as the output field is not a true input to the operation. When \(\text{isPre}\) is true, the pre-order traversal allows to satisfy the ancestor dependence.

Lst. 6.4 is an example of a recursive specification of an operation, which computes the value of the output field at label \(l\) as sum given by \(f_1[l] + \text{output}[l.\text{child0}] + \text{output}[l.\text{child0}]\). This is an example where \(\text{isPre}\) is false. When \(\text{isPre}\) is false, the post-order traversal allows to satisfy descendent dependence. The dependence is the result of DR data access to output field. In this case \(\text{output}[l]\) requires the value of \(\text{output}[l.\text{child0}]\) and \(\text{output}[l.\text{child1}]\).
Listing 6.3: Primitive operator that accumulates coefficients from parent nodes

```c
// ST is the type for spatial tree, res is output, f1 is input
// same node access for f1
#pragma f1:SNR
// ancestor access for output
#pragma output:AR
void OpCP(Label l, ST output, ST f1) {
    bool r = false;
    // pre-Compute
    computeCP(l, output, f1);
    // need Recursion
    r = needRecursion(l, output, f1);
    if (r) {
        OpCP(l.child0, output, f1);
        OpCP(l.child1, output, f1);
    }
    // post-compute
    computeCP(l, output, f1);
}

bool needRecursion(Label l, ST output, ST f1) {
    return !output[l].is_leaf;
}

void computeCP(Label l, ST output, ST f1) {
    double cf = f1[l].coeff;
    if (l == root) output[l].coeff = cf;
    else output[l].coeff = cf + output[l.parent].coeff;
    output[l].is_leaf = f1[l].is_leaf;
}
```

Listing 6.4: Primitive operator that accumulates coefficients from children nodes

```c
#pragma f1:SNR // same node access for f1
#pragma output:DR // descendent access output
void OpCC(Label l, ST output, ST f1) {
    bool r = false;
    // need Recursion
    r = needRecursionCC(l, output, f1);
    if (r) {
        OpCC(l.child0, res, f1);
        OpCC(l.child1, res, f1);
    }
    computeCC(l, output, f1);
    /* post-compute */
}

bool needRecursionCC(Label l, ST output, ST f1) {
    return !f1[l].is_leaf;
}

void computeCC(Label l, ST output, ST f1) {
    double cf = f1[l].coeff;
    if (f1[l].is_leaf) output[l].coeff = cf;
    else output[l].coeff = cf + output[l.child0].coeff
        + output[l.child1].coeff;
    output[l].is_leaf = f1[l].is_leaf;
}
```
Listing 6.5: Sequence of primitive operators in FuseT

1) SpatialTree f1=GenerateRandomSpatialTree();
2) SpatialTree i1,output;
3) OpCP(root,i1,f1);
4) OpCC(root,output,i1);

Lst. 6.5 shows a program written using operators specified in Lst. 6.3 and Lst. 6.4. In this example, OpCP computes i1 which is used as an input to OpCC. We use this example to motivate fusion in the next section.

6.4 Fusion in FuseT

In FuseT, the fusion of two or more operators is the interleaving of their node operations with the intent of improving data locality. Consider a sequence of primitive operations where result of an operator acts as an operand to one or more operations in the sequence. Without fusion, each intermediate spatial tree is fully traversed when it is produced, and then fully traversed each time it is used as an input operand of an operator later in the sequence. If the intermediate spatial trees are larger than cache, they will require main memory access for each operator that produces/consumes each intermediate. By fusing node operations across operators, significant reduction in memory traffic can be achieved.

In the Lst. 6.5 example, i1 is produced fully by OpCP, before it is used by OpCC. If i1 is larger than cache, then it must be stored in memory, and OpCC must read it from memory. By fusing the node operations between these two operators, memory access for i1 can be greatly reduced.

Lst. 6.6, shows such a fused schedule. In this schedule, notice that the recursion ends when i1[1].is_leaf is true. Thus at these node labels, i1 is immediately
consumed by computeCC. In fact i1 produced at label l will still be in cache when computeCC reads it, if its descendent sub-tree fits in cache. A significant fraction of the memory access is avoided by reading i1 from cache for all the node labels where the descendent tree fits in cache.

In this section, we describe a recursive specification for fused schedule that allows interleaving of node operations in the computation space of a FuseT program to improve locality.

### 6.4.1 The Computation Space of a FuseT Program

A program in FuseT can consist of a sequence of operator applications, forming in an operator DAG (OpDAG) with k operators. Consider a program that computes in 1D space. The spatial tree in 1D is a binary partition tree. Let d be the depth of the deepest binary partition tree in the program. We can think of a full binary partition tree of depth d as the 'canonical' spatial tree for this program, with a total of $2^d - 1$ nodes.
nodes. For such a program the total computational space is a subset of $k \times 2^d - 1$ computational points. Each point in this space can be written as $\text{op}(i, j)$, which represents the node operation of operator labeled $i$, at node label $j$. In other words, $\text{op}(i, j)$ is computeOP($j, \ldots$), in the recursive specification for operator labeled $i$, which produces the value of output field for operator $i$ at node label $j$. We can use $\text{op}(i, *)$ to represent the application of operator $i$ to produce the entire output tree.

### 6.4.2 Recursive Specification of a Fused Schedule

For a sequence of $k$ operators, there is an exponentially large number of possible schedules for interleaving node operations given by $(k \times (2^d - 1))!$. We consider a subspace of these schedules which allows for a recursive specification, just as the primitive operators, while affording locality benefits. We define the fused schedule for an operation DAG as follows:

**Definition 6.4.1.** For an operator DAG, a fused schedule for computing the DAG is a schedule with recursive specification given by Lst. 6.7, where the true input fields to the DAG are passed in as InputList parameters, and the output fields are passed in as OutputList parameters. In this recursive schedule, a node operation $\text{op}(i, j)$ is executed either by $\text{preFused}(j, \ldots)$ or by $\text{postFused}(j, \ldots)$.

The fused schedule given by this recursive specification is more general than Lst. 6.2, as it allows for writing data to multiple fields within a single tree traversal. The assignment to values of a field at label $l$ happens exactly once, either by $\text{preFused}(1, \ldots)$, or by $\text{postFused}(1, \ldots)$ in the fused schedule. Let $U[l]$ be set of all the node operations given by $\text{op}(*, l)$. Let $P[l]$ be the set of all the node operations executed by $\text{preFused}(1, \ldots)$. Let $F[l]$ be the set of all the node operations
Listing 6.7: Recursive specification for fused operators

```c
FusedOp(Label l, OutputList ol, InputList pl) {
    bool r = false;
    preFused(l, ol, pl);
    r = needRecursionFused(l, ol, pl);
    if (r) {
        FusedOp(l.child0, ol, nol);
        FusedOp(l.child1, ol, nol);
    }
    postFused(l, ol, pl);
}
```

executed by `postFused(l,..)`. Then, by Def. 6.4.1 we have:

\[
P[l] \cup F[l] = U[l] \quad (6.3) \\
P[l] \cap F[l] = \emptyset \quad (6.4)
\]

6.4.3 Ordering between Node Operations

Here we establish an ordering between different node operations in a fused schedule. The ordering relations are important for dependence analysis we present in Sec. 6.5, to identify OpDAGs within a program that can be fused.

**Lemma 6.4.2.** In a fused schedule, \(P[l] \prec U[l.descendent] \prec F[l]\), meaning that node operations in \(P[l]\) precedes any node operation in \(U[l.descendent]\), which in turn precedes any node operation in \(F[l]\).

**Proof.** The lemma is true by construction. In the recursive specification of fused schedule given by Lst. 6.7, call to `preFused` at a node label \(l\) precedes the recursive calls to the child nodes of \(l\), which precedes `postFused`. These calls are responsible for executing operations in \(P[l]\), \(U[l.descendent]\) and \(F[l]\), respectively. Therefore, \(P[l] \prec U[l.descendent] \prec F[l]\).

**Corollary 6.4.3.** In a fused schedule, \(P[l.ancestor] \prec U[l] \prec F[l.ancestor]\).
Proof. A node \( l \) is an ancestor to node \( l\text{.descendent} \). Thus, if \( P[l] \prec U[l\text{.descendent}] \), then we get \( P[l\text{.ancestor}] \prec U[l] \) by renaming \( l\text{.descendent} \) as \( l \).

For the next lemma, let \( l\text{.s} \) represent all siblings of a node \( l \), and \( l\text{.d} \) represent descendents of a node \( l \).

**Lemma 6.4.4.** In a fused schedule if \( UC[l] = U[l] \cup U[l\text{.d}] \), \( UC[l\text{.s}] = U[l\text{.s}] \cup U[l\text{.s.d}] \), and \( n_1 \in UC[l] \land n_2 \in UC[l\text{.s}] \), then the order of execution between \( n_1 \) and \( n_2 \) is non-deterministic. We represent this non-determinism as \( UC[l] \not\prec \not\succ UC[l\text{.s}] \).

Proof. By Construction: \( Op(l,..) \) and \( Op(l\text{.s},..) \) are executed in parallel, thus the order between them is non-deterministic. \( Op(l,..) \) executes \( UC[l] = U[l] \cup U[l\text{.d}] \), and \( Op(l\text{.s},..) \) executes \( UC[l\text{.s}] = U[l\text{.s}] \cup U[l\text{.s.d}] \). So, ordering between any pair of \( n_1, n_2 \mid n_1 \in UC[l] \land n_2 \in UC[l\text{.s}] \), is non-deterministic.

**Corollary 6.4.5.** In a fused schedule, if \( l \) and \( m \) are two unique nodes that are related by SSR, then \( U[l] \not\prec \not\succ U[m] \), meaning that the ordering between any two node operations in \( U[l] \) and \( U[m] \), respectively is non-deterministic.

Proof. By Contradiction: Because \( l \) and \( m \) are related by SSR, \( l \) is not an ancestor of \( m \), and vice versa. If \( c \) is the first common ancestor of \( l \) and \( m \), then \( l \) and \( m \) are either different successors of \( c \) or, descendents of different successors of \( c \). Therefore, \( U[l] \subset UC[c\text{.child}0] \) and \( U[m] \subset UC[c\text{.child}1] \). Contradiction: \( UC[c\text{.child}0] \not\prec \not\succ UC[c\text{.child}1] \) according to Lemma 6.4.4. Thus, \( U[l] \not\prec \not\succ U[m] \).

### 6.5 When Does a Valid Fused Schedule Exist?

In this section we establish the necessary and sufficient conditions for a sequence of operations to have valid fused schedule. These are conditions on the chain of
dependences in the producer consumer paths between operators in the sequence. To identify these conditions, we first classify dependences between operators.

### 6.5.1 Dependences in Operator DAG

In an OpDAG, a dependence exists between two operators if the output field produced by the first operator is an operand to the second operator. We call such pair of operators a producer-consumer pair. Consider a producer operator $Op\text{Producer}$ that produces an output field $out_p$, and a consumer operator $Op\text{Consumer}$ that consumes $out_p$. Then, the dependence between $Op\text{Producer}$ and $Op\text{Consumer}$ can be classified into four mutually exclusive categories:

1. Same Node Dependence (SND or $\psi$)
2. Ancestor Dependence (AD or $\alpha$)
3. Descendent Dependence (DD or $\delta$)
4. Some Sibling Dependence (SSD or $\sigma$)

For example, the dependence between $Op\text{Consumer}$ and $Op\text{Producer}$ is called SND if the data access type between $Op\text{Consumer}$ and $out_p$ is of type SNR. AD, DD and SSD are defined similarly. There can be more than one type of dependence between $Op\text{Consumer}$ and $Op\text{Producer}$, if the data access between $Op\text{Consumer}$ and $out_p$ is of more than one type.

We will use the notation $Op_2 \leftarrow^x Op_1$, to imply that there is data flow dependence between $Op_2$ and $Op_1$, where $Op_2$ is the consumer and $Op_1$ is the producer, and the type of dependence is given by $x$. For example, $Op_2 \leftarrow^{\psi,\delta} Op_1$ implies Same Node
Figure 6.2: This figure shows an OpDAG representing the sequence of operations given by Lst. 6.5. The blue arrows (labeled AD, SND and DD) shows dependences between OpCP and OpCC, and the green arrows (labeled f1, i1 and Output) shows the input and output fields to each of the operators.

Dependence ($\psi$) and Descendent Dependence ($\delta$) between the two operators, where $Op_2$ is the consumer and $Op_1$ is the producer.

6.5.2 Necessary and Sufficient Conditions

In an OpDAG, an edge exists between two operator nodes, $Op_1$ to $Op_2$, if $Op_2 \leftarrow^x Op_1$, where $x \in \{\psi, \alpha, \delta, \sigma\}$ is the dependence type between the operators corresponding to the graph nodes. Fig. 6.2 shows the OpDAG for the program given in Lst. 6.5.

Now, consider a pair of operators in an OpDAG given by $Op_j$ and $Op_{j-k}$. Let

$$Op_j \leftarrow^{x_j} Op_{j-1} \leftarrow \ldots Op_{j-k+1} \leftarrow^{x_{j-k+1}} Op_{j-k}$$

(6.5)

represent a path between these two operators in the OpDAG, where $x_j$ is the type of dependence between $Op_j$ and $Op_{j-1}$. We can represent the sequence of dependences (DS) in this path as

$$x_jx_{j-1}x_{j-2} \ldots x_{j-k+1}$$

(6.6)

171
where $x_{j-s} \in \{\psi, \alpha, \delta, \sigma\}$. The necessary and sufficient conditions on the sequence of dependences (DS) for an OpDAG to have a valid schedule is given by Theorem 6.5.1.

**Theorem 6.5.1.** An OpDAG has a valid fused schedule iff the sequence of dependences in all paths between any two operators in the OpDAG can be recognized by the regular expression $R_{\text{valid}} : (\delta^* \psi^*)^* (\alpha^* \psi^*)^*$. We refer to such an OpDAG as a valid-OpDAG.

Theorem 6.5.1 requires that any sequence of dependences in the OpDAG cannot have $\sigma$ dependence, and all the $\alpha$ dependences must precede all the $\delta$ dependences. We will refer to the set of all the dependence sequences recognizable by $R_{\text{valid}}$ as $V$. We will refer to set of all possible sequences as $U$.

In the rest of this section we prove this theorem in two parts. First, we prove the necessary condition by showing that any sequence of dependences in $U - V$ does not have a valid fused schedule. Therefore, all dependence sequences in an OpDAG must be in $V$ if there exists a valid schedule as $V$ and $U - V$ are mutually exclusive. Next, we prove the sufficient condition by showing that valid fused schedule can be constructed for any OpDAG where all dependence sequences are in $V$.

### 6.5.3 Necessary Condition

We prove the necessary condition by showing that a pair of operators $Op_i$ and $Op_j$ in an OpDAG cannot be fused, if the dependence sequence between them is in $U - V$. We show this in two parts by splitting $U - V$ into two disjoint sets $N$ and $C$, and showing that any sequence of operators with dependence sequence either in $N$ or $C$ does not have a valid fused schedule.
Table 6.1: This table shows sets representing sequence of dependences (DS), and regular expressions for recognizing them.

<table>
<thead>
<tr>
<th>Sets of DS</th>
<th>Regular Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>$(\sigma^<em>\psi^</em>\alpha^<em>\delta^</em>)^*$</td>
</tr>
<tr>
<td>V</td>
<td>$(\delta^<em>\psi^</em>)^<em>(\alpha^</em>\psi^<em>)^</em>$</td>
</tr>
<tr>
<td>M</td>
<td>$(\psi^<em>\alpha^</em>\delta^<em>)^</em>$</td>
</tr>
<tr>
<td>N=U-M</td>
<td>$(\sigma^<em>\psi^</em>\alpha^<em>\delta^</em>)^<em>\sigma^+(\sigma^</em>\psi^<em>\alpha^</em>\delta^<em>)^</em>$</td>
</tr>
<tr>
<td>C=M-V</td>
<td>$(\delta^<em>\psi^</em>\alpha^<em>)^</em>\alpha\psi^<em>\delta^</em>(\delta^<em>\psi^</em>\alpha^<em>)^</em>$</td>
</tr>
</tbody>
</table>

Consider the set of all dependence sequences $M$ that can be recognized by $(\psi^*\alpha^*\delta^*)^*$. Notice that $M \subset U$ and $V \subset M$. We can rewrite $U - V$ as,

$$U - V = (U - M) + (M - V) \quad (6.7)$$

$$= N + C \quad (6.8)$$

where $N = U - M$ and $C = M - V$. Notice that $N, C$ and $V$ are disjoint sets and $U = N + C + V$. $N$ represents all dependence sequences with at least one $\sigma$ dependence, and $C$ represents all the dependence sequences without any $\sigma$ dependence. In addition, $C$ has at least one $\alpha$ and one $\delta$ dependence in the sequence such that $\delta$ precedes $\alpha$.

Table 6.1, shows the regular expressions representing these sets.

**Lemma 6.5.2.** $Op_i$ and $Op_j$ cannot be fused if there exists a path between them such that $Op_s \leftarrow^\sigma Op_t$, where $Op_s$ and $Op_t$ are two operators in the path.

**Proof.** By Contradiction: A fused schedule that computes $Op_i$ and $Op_j$, must also compute all operators in the path between them. Thus, the fused schedule must also compute $Op_s$ and $Op_t$. For this fused schedule to be valid, $Op_s \leftarrow^\sigma Op_t$ dependence must be satisfied. The dependence is satisfied if $op(t, l) \prec op(s, l')$, where $l$ and $l'$ are
related by SSR. Contradiction: \( op(t, l) \in U[l] \) and \( op(s, l') \in U[l'] \). But \( U[l] \not\prec U[l'] \) according to Corollary 6.4.5.

**Corollary 6.5.3.** An OpDAG does not have a valid fused schedule if there exist a dependence sequence between any pair of operators in the OpDAG is in \( N \).

**Proof.** There exists at least a pair of nodes in the OpDAG, such that the dependence sequence between them has a \( \sigma \) dependence because at least one of the dependence sequence in the OpDAG is in \( N \). Thus, by Lemma 6.5.2, the OpDAG cannot have a valid fused schedule.

**Lemma 6.5.4.** \( Op_i \) and \( Op_j \) cannot be fused if there exists a sub-path \( Op_{s-1} \leftarrow^\alpha \ldots Op_{s-k+1} \leftarrow^\delta Op_{s-k} \) between them, where the sub-path has dependence sequence given by \( \alpha \psi^* \delta \).

**Proof.** By Contradiction: For a valid fused schedule, the following three conditions must be true:

I) \( op(s - k, \text{descendant}) \) must precede \( op(s - k + 1, l) \) in the fused schedule to satisfy \( \delta \) dependence. \( op(s - k + 1, \text{descendant}) \in U[l, \text{descendant}] \) and \( P[l] \prec U[l, \text{descendant}] \), according to Lemma. 6.4.2. Thus, \( op(s - k + 1, l) \notin P[l] \rightarrow op(s - k + 1, l) \in F[l] \).

II) To satisfy \( \psi^* \), \( op(s - k + 1, l) \prec op(s - k + 2, l) \prec \ldots \prec op(s - 1, l) \) in a valid fused schedule. \( op(s - k + 1, l) \) is in \( F[l] \) (from i), and \( P[l] \prec F[l] \) according to Lemma. 6.4.2. Thus, \( op(s - 1) \notin P[l] \rightarrow op(s - 1, l) \in F[l] \).

III) \( op(s - 1, \text{ancestor}) \) must precede \( op(s, l) \), to satisfy the \( \alpha \) dependence. \( op(s, l) \in U[l] \), and \( U[l] \prec F[l, \text{ancestor}] \), according to Corollary. 6.4.3. Thus, \( op(s - 1, \text{ancestor}) \notin F[l] \).
Corollary 6.5.5. An OpDAG does not have a valid fused schedule if the dependence sequence between any pair of operators in the OpDAG is in $C$.

Proof. There exists at least a pair of nodes in the OpDAG, such that the dependence sequence between them has a $\delta$ dependence followed by an $\alpha$ dependence because the OpDAG is in $C$, i.e it can be recognized using $\alpha\psi\delta$. Thus, by Lemma 6.5.4, the OpDAG cannot have a valid fused schedule.

Corollary 6.5.6. A valid fused schedule does not exist if the dependence sequence between any pair of operators in the OpDAG is in $U - V$. In other words, if there exist a valid fused schedule, all dependence sequences in OpDAG must be in $V$.

Proof. $U - V = N + C$, thus by Corollary 6.5.3 and Corollary 6.5.5, a fused schedule cannot exist if at least a one dependence sequence is in $U - V$. Thus a valid schedule cannot have any dependence sequence in $U - V$. Therefore, it must all be in $V$.

6.5.4 Sufficient Condition

Definition 6.5.7. Valid fused Schedule: A schedule given by Lst. 6.7 is a valid fused schedule for an OpDAG if it satisfies all dependences in the OpDAG.

We prove that validity of an OpDAG is a sufficient condition for a valid fused schedule to exist in two steps. i) We identify sufficient conditions for a fused schedule to satisfy all dependences in a valid OpDAG (in this section). ii) We present an algorithm for constructing a fused schedule that satisfies all the sufficient conditions.
(in the next section). Thus, the schedule produced by our algorithm will be valid by construction.

Sufficient conditions to satisfy all dependences in a valid OpDAG is simply the union of all conditions required to satisfy individual sequence of dependences (DS) in the valid OpDAG. To find such conditions, we partition dependence sequences (DS) in a valid-OpDAG into three sub-sequence. For each sub-sequence we identify sufficient conditions to satisfy all dependences in that sub-sequence. A DS in a valid-OpDAG can be expressed using a regular expression given by

\[ R_{\text{valid}} : (\delta^*\psi^*)^* (\alpha^*\psi^*)^* \]  \hspace{1cm} (6.9)

All the \( \alpha \) dependences precede all the \( \delta \) dependences for a DS in a valid-OpDAG. Without loss of generality, if we assume that a DS contains at least one \( \alpha \) and at least one \( \delta \) dependence, we can re-write \( R_{\text{valid}} \) for this DS as

\[ N_{\text{valid}} : D_p S_p A_p \]  \hspace{1cm} (6.10)

\[ D_p : (\delta^*\psi^*)^* \delta \]  \hspace{1cm} (6.11)

\[ S_p : \psi^* \]  \hspace{1cm} (6.12)

\[ A_p : \alpha (\alpha^*\psi^*)^* \]  \hspace{1cm} (6.13)

Next we identify sufficient conditions to satisfy \( A_p \), \( S_p \) given \( A_p \) is satisfied, and finally \( D_p \), given \( A_p \) and \( S_p \) are satisfied.

**Sufficient Conditions for \( A_p \)**

We show that if there exists a path from \( Op_{j-k} \) to \( Op_j \) such that the DS for the path is given by \( A_p \), then placing all operators in that path in \( P[l] \) and ordering them in the order of appearance in the path is sufficient to satisfy \( A_p \) (proved in
Corollary. 6.5.10). We prove this sufficient condition with the help of Lemma. 6.5.8 and Lemma. 6.4.2.

Lemma 6.5.8. If $Op_i$ and $Op_j$ are two operators labeled $i$ and $j$ in an OpDAG such that $Op_i \leftarrow^\alpha Op_j$, then the dependence is satisfied iff $op(j,l) \in P[l]$ in the fused schedule.

Proof. $Op_i \leftarrow^\alpha Op_j$ implies that in a valid schedule

$op(j,l.ancestor) \prec op(i,l)$. From Eq.6.3 and Eq. 6.4, either $op(j,l) \in P[l]$ or $op(j,l) \in F[l]$ but not both. If $op(j,l) \in P[l]$, then from Corollary 6.4.3, the dependence is satisfied because $op(j,l.ancestor) \in P[l.ancestor]$ and $op[i,l] \in U[l]$. If $op(j,l) \in F[l]$, then $op(i,l) \prec op(j,l.ancestor)$, violating the dependence. Thus if the dependence is satisfied, then $op(j,l). \in P[l]$, and vice versa.

Lemma 6.5.9. In a valid OpDAG, if $Op_i \leftarrow^\psi Op_j \land op(i,l) \in P[l]$ then the dependence is satisfied if, $op(j,l) \in P[l]$ and $op(j,l)$ is executed before $op(i,l)$ in preFused.

Proof. $op(j,l) \prec op(i,l)$ must be true to satisfy the dependence. If $op(j,l)$ and $op(i,l)$ are both in $P[l]$, they are executed by preFused(l,..). If preFused(l,..) executes $op(j,l)$ before $op(i,l)$, then $op(j,l) \prec op(i,l)$, thus satisfying the dependence.

Corollary 6.5.10. If $Op_j, Op_{j-1}, \ldots, Op_{j-k+1}, Op_{j-k}$ is a chain of operators in valid OpDAG such that $Op_j \leftarrow^\alpha Op_{j-1}$, then in fused schedule satisfies all the dependences in the path if, $op(j-k+s,l) \in P[l] \mid 0 \leq s < k, s \in Z$, and the operators are executed by preFused in the same order as in the path.

Proof. By Induction: Because OpDAG is valid, all the dependences in the chain can be expressed by $A_p$. Therefore, $Op_{j-k+s+1} \leftarrow^x Op_{j-k+s} \mid x \in \alpha, \psi$. 177
**Inductive Case:** If \( x = \alpha \), or \( x = \psi \land op(j - k + s + 1, l) \in P[l] \), then a fused schedule satisfies dependence \( x \) if \( op(j - k + s, l) \in P[l] \) and \( op(j - k + s, l) \) executes before \( op(j - k + s + 1, l) \) in \( \text{preFused} \) according to Lemma 6.5.8 and Lemma 6.5.9, respectively.

**Base Case:** If \( op(j - 1, l) \in P[l] \) then \( Op_j \leftarrow_{AD} Op_{j-1} \) is satisfied by Lemma 6.5.8.

Thus, by induction if \( op(j - 1, l), op(j - 2, l), \ldots, op(j - k + s, l) \in P[l] \) in the order of appearance in the path, then all the dependence in \( DS \) is satisfied.

**Sufficient Conditions for \( S_p \)**

**Lemma 6.5.11.** If there exists a path from \( Op_{j-k} \) to \( Op_j \) such that the \( DS \) for the path is given by \( S_p \), then all the dependences in the path is satisfied regardless of the placement of the operators in \( P[l] \) or \( F[l] \), if operators are executed by either \( \text{preFused} \), or \( \text{postFused} \) in the order that they appear in the path.

*Proof.* Trivially true by the definition of dependence satisfaction, i.e. the dependences are satisfied if \( op(j - s, l) \prec op(j - s + 1, l) \mid 0 < s \leq k \land s \in \mathbb{Z} \). This condition allows the operators to be in either \( P[l] \) or \( F[l] \), as long as the order of execution is maintained.

**Sufficient Conditions for \( F_p \)**

We show that if there exists a path from \( Op_{j-k} \) to \( Op_j \) such that the \( DS \) for the path is given by \( D_p \), then placing all operators in that path in \( F[l] \) and ordering them in the order of appearance in the path is sufficient to satisfy \( D_p \) (proved in Corollary. 6.5.14).
Lemma 6.5.12. If $Op_i$ and $Op_j$ are two operators labeled $i$ and $j$ in OpDAG such that $Op_i \leftarrow^\delta Op_j$, then a valid fused schedule for OpDAG given by Lst. 6.7, $op(i,l) \in F[l]$.

Proof. $Op_i \leftarrow^\delta Op_j$ implies that in a valid schedule $op(j, l.descendent) \prec op(i,l)$. From Eq.6.3 and Eq. 6.4, either $op(i,l) \in P[l]$ or $op(i,l) \in F[l]$ but not both. If $op(i,l) \in F[l]$, then from Lemma 6.4.2, the dependence is satisfied because $op(i,l) \in F[l]$ and $op[j,l.descendent] \in U[l.descendent]$. If $op(i,l) \in P[l]$, then $op(i,l) \prec op(j,l.descendent)$, violating the dependence. Thus if the dependence is satisfied, then $op(i,l) \in F[l]$, and vice versa.

Lemma 6.5.13. In a valid OpDAG, if $Op_i \leftarrow^\psi Op_j \wedge op(j,l) \in F[l]$ then the dependence is satisfied if, $op(i,l) \in F[l]$ and $op(j,l) \in F[l]$ is executed before $op(i,l)$ in postFused.

Proof. $op(j,l) \prec op(i,l)$ must be true to satisfy the dependence. If $op(j,l)$ and $op(i,l)$ are both in $F[l]$, they are executed by $postFused(1, \ldots)$. If $postFused(1, \ldots)$ executes $op(j,l)$ before $op(i,l)$, then $op(j,l) \prec op(i,l)$, thus satisfying the dependence.

Corollary 6.5.14. If $Op_j, Op_{j-1}, \ldots, Op_{j-k+1}, Op_{j-k}$ is a chain of operators in valid OpDAG such that $Op_{j-k+1} \leftarrow^\delta Op_{j-k}$, then a fused schedule satisfies all the dependences in the path if, $op(j - k + s, l) \in F[l] \mid 0 < s \leq k, s \in \mathbb{Z}$, and the operators are executed by $postFused$ in the same order as in the path.

Proof. By Induction: Because OpDAG is valid, all the dependences in the chain can be expressed by $D_p$. Therefore, $Op_{j-k+s+1} \leftarrow^x Op_{j-k+s} \mid x \in \delta, \psi$. 

179
**Listing 6.8:** Identifying pre- and post-order operators in valid OpDAG

```cpp
vector<OpNode> preNodes;
vector<OpNode> postNodes;

void getComputeFused(OperatorDAG opDAG) {
  for (op : opDAG) op.visited = false;
  for (op : opDAG)
    if (!op.visited && isPreOp(op)) addPreOps(op);
  for (op : opDAG) op.visited = false;
  for (op : opDAG)
    if (!op.visited && isPostOp(op)) addPostOps(op);
  for (op : opDAG) {
    if (op.isPost && op.isPre) throw "Cannot fuse OpDAG";
    if (!op.isPre && !op.isPost) preNodes.push(op);}
  //sort the nodes based on the ordering in input program
  sort(preNodes); sort(postNodes);
}

bool isPreOp(OpNode op) {
  for (cn : op.consumers)
    if (cn <-AD op) return true;
  return false; }

bool isPostOp(OpNode op) {
  for (pn : op.producer)
    if (op <-DD pn) return true;
  return false; }

void addPreOps(OpNode op) {
  if (!op.visited) {
    op.visited=true; op.isPre=true; preNodes.push(op);
    for (pn : op.producer) addPreOps(pn);} }

void addPostOps(OpNode op) {
  if (!op.visited) {
    op.visited=true; op.isPost=true; postNodes.push(op);
    for (cn : op.consumers) addPostOps(cn);}}
```

**Inductive Case:** In a valid fused schedule, if \( x = \delta \), or \( x = \psi \land op(j - k + s, l) \in F[l] \), then \( op(j - k + s + 1, l) \in F[l] \), and \( op(j - k + s + 1, l) \) is executed after \( op(j - k + s, l) \) in postFused according to Lemma 6.5.12 and Lemma 6.5.13, respectively.

**Base Case:** \( op(j - k + 1, l) \in F[l] \) by Lemma 6.5.12.

Thus, by induction we have \( op(j - k + 1, l), op(j - k + 2, l), \ldots, op(j - k + s, l) \in F[l] \), in the order than it appears in the path.  

---

180
6.6 Traversal Fusion

In this section, we present our approach to fusing the operators in a given DAG. We begin with an algorithm to determine the fused schedule given a valid OpDAG (as defined in Theorem 6.5.1). Then, we present an algorithm to partition a given program written in FuseT into a dependence-preserving sequence of valid OpDAGs.

6.6.1 Generating a Fused Schedule for a Valid OpDAG

Computing a schedule for a valid OpDAG involves classifying operators to be executed in either pre- and post-order phase of the fused traversal and determining the relative execution order for the statements of operators in the same phase, such that the dependences are preserved. Building on Corollaries 6.5.10 and 6.5.14 and Lemma 6.5.11, the getComputeFused() routine in Lst. 6.8 shows the algorithm to compute a schedule for a given valid OpDAG.

Given a node in the OpDAG, the algorithm checks to see if it is constrained to be in the pre-order or post-order phase by inspecting its consumers and producers,
respectively (Lines 6 and 10). A node whose consumer depends on it with a $\alpha$ dependence is marked to be in the pre-order phase (Lemma 6.5.8), marked to be in the post-order phase (Lemma 6.5.12) if it has a $\delta$ dependence. Given a marked node, the algorithm marks all transitive producers (consumers) of a node marked to be in the pre-order (post-order) phase to also be in the pre-(post-)order phase. Nodes that have not been constrained to execute in either tree traversal phase can be executed in either phase. The algorithm assigns them to the pre-order phase (Line 16).

At this point, all nodes have been assigned to preNodes or postNodes, designating the tree traversal phase they belong to. Within each phase, the nodes are ordered based on their appearance in the input program. Based on Corollaries 6.5.10 and 6.5.14 and Lemma 6.5.11, we know that this generates a dependence-preserving schedule.

The fused operator for a valid OpDAG is shown in Lst. 6.9. A binary tree is shown for illustration. For a given label, the operator first executes all operations in the pre-order phase and updates the termination condition for all operators. This is followed by the recursive call and execution of the operators in the post-order phase.

**Illustration** Lst. 6.6 shows a valid fused schedule for the operator sequence—OpCP and OpCC—in Lst. 6.5. The corresponding dependences are: $OpCP \leftarrow^\alpha OpCP$, $OpCC \leftarrow^\psi OpCP$ and $OpCC \leftarrow^\delta OpCC$. The OpDAG and the dependence information are shown in Fig. 6.2.

### 6.6.2 Generating a Sequence of Valid OpDAGs

If an OpDAG has a dependence sequence that is not in $V$, then it cannot be fused. However, an OpDAG can be decomposed into sub-DAGs, each of which constitutes
a valid fusible OpDAG. There can be more than one sequence of valid OpDAGs to compute the original operator DAG, each with an associated cost.

**Definition 6.6.1.** The cost to compute an operator using a sequence of valid OpDAGs is the sum of the number of true inputs to each of the valid OpDAGs in the sequence.

Def. 6.6.1 defines the cost of a sequence of valid OpDAGs, in terms of the relative volume of data that must be read from slow memory to fast memory by it. The underlying assumptions behind this definition are three-fold:

1. The total volume of tree data produced by each operator in the OpDAG is larger than the size of fast memory,

2. the total volume of tree field data produced by each operator is approximately the same, and

3. if a producer operator A and a consumer operator B are part of a single valid OpDAG, then data at a tree node produced by A is consumed by B while it is still in fast memory.

On the other hand, if a producer operator A and consumer operator B are not in the same valid OpDAG, the entire tree data produced by A must be read back into fast memory when B is executed. Thus, the number of trees that must be read from slow memory corresponds to the sum of number of true inputs to all the valid OpDAGs that together compute the given operator dag.
Listing 6.10: Generating a valid OpDAG partitions

1 //returns a dependence-preserving sequence of sub-dags that are valid OpDAGs
2 vector<OpDAG> getSequence(OpDAG odag) {
3 vector<OpDAG> result, t0,t1;
4 //returns largest partition without some sibling dependence
5 OpDAG sp = getSigmaPartition(odag);
6 //returns largest partition without delta followed
7 //by alpha dependence
8 OpDAG adp = getAlphaDeltaPartition(sp);
9 result.push(adp); //recursively compute remaining partitions
10 result.push(getSequence(sp - adp));
11 result.push(getSequence(odag - sp));
12 return result; }

13
14 list<vertices> getDTList(OpDAG opdag, vertex s, 
15 dependencyType dt) {
16 list<vertices> result.push(s);
17 queue<vertices> q.push(s);
18 while(!q.isEmpty()) {
19 s = q.deque(); s.visited=true;
20 /* neighbors: producers or consumers*/
21 for (c : s.neighbors) {
22 Edge e = getEdge(s,c);
23 if(e.type=='dt' && !c.type[dt]) {
24 c.type[dt]=true; result.push(c);
25 } else if(!c.visited && !c.type[dt]) q.enqueue(c); }
26 return result; }

27
28 //returns the largest partition without delta followed by
29 alpha dependence*/
30 OpDAG getAlphaDeltaPartition(OpDAG odag) {
31 list<vertices> source = /*odag's true input vertices*/
32 //merges all vertices in source
33 OpDAG new_odag = sourceMergedOpDAG(odag,source);
34 list<vertices> delta_list;
35 delta_list=getDTList(new_odag,new_odag.source,''delta'');
36 for (op : new_odag) op.visited = false;
37 for (d : delta_list) {
38 list<vertices> alpha_list = getDTList(new_odag,d,''alpha'');
39 for (a : alpha_list) {
40 //returns all vertices reachable from 'a'
41 list<vertices> reachable_from_a =
42 getReachable(new_odag,a);
43 new_odag.remove(reachable_from_a);}
44 return new_odag; }

45 //returns the largest partition without some sibling dependence
46 OpDAG getSigmaPartition(OpDAG odag) {
47 list<vertices> source = /*odag's true input vertices*/
48 //merges all vertices in the source
49 OpDAG new_odag = sourceMergedOpDAG(odag,source);
50 list<vertices> sigma_list;
51 sigma_list=getDTList(new_odag,new_odag.source,''sigma'');
52 for (s : sigma_list) {
53 //returns all vertices reachable from 's'
54 list<vertices> reachable_from_s = getReachable(new_odag,s);
55 new_odag.remove(reachable_from_s);}
56 return new_odag; }

184
It turns out that finding the optimal sequence of valid OpDAGs that minimizes the cost is an NP-hard problem. We present a heuristic approach based on a greedy algorithm, that runs in $O(P(|V|+|E|))$ time, where $P$ is the number of valid OpDAGs in the sequence, $|V|$ and $|E|$ are the number of vertices and edges in the operator dag, respectively. The algorithm, shown in Lst. 6.10, works by recursively finding the largest convex partition that can be fused.

Lst. 6.10 shows the algorithm for finding a sequence of valid OpDAGs using the aforementioned greedy algorithm. It starts by finding the largest convex partition (LCP) that can be fused such that all the inputs to this LCP are true inputs. This is done in two steps in getSequence: (i) First it finds the LCP (stored in $sp$) that does not contain any $\sigma$ dependence. (ii) It finds the LCP (stored in $adp$) within $sp$ without any $\delta\psi^*\alpha$ DS. $adp$ is the first valid OpDAG. The algorithm then recursively computes the sequence of largest valid-opDAGs in $sp-adp$ and $adag-sp$.

The complexity of algorithm in Lst. 6.10 is $O(P(|V|+|E|))$. Each valid OpDAG requires a single execution of getAlphaDeltaPartition and getSigmaPartition. The complexity of each procedure is $O(|V| + |E|)$. Notice that getDTList runs in $O(|V| + |E|)$ time. In getSigmaPartition, the for loop can visit each node and each edge in $new_{odag}$ at most once, as they are removed the first time they are encountered. Similarly, the nested loop in getAlphaDeltaPartition visits each vertex and edge in $new_{odag}$ at most once, as they are either marked visited by getDTList in the outer-loop, or they are removed by the inner loop. Thus, the complexity of computing each valid OpDAG is $O(|V| + |E|)$, and the complexity of the entire algorithm is $O(P(|V| + |E|))$. 

185
Figure 6.3: a) The figure on the left shows the perf. improvement of fused Sum benchmark over the baseline for a varying length of the input vector. b) The figure on the right compares scalability between fused and baseline versions of Sum and OuterSum on up to 16 cores.
6.7 Evaluation

In this section, we demonstrate the performance improvement that can be attained using fusion for tree traversal applications. Specifically, we present performance improvements in a k-d tree based framework called MADNESS (Multiresolution Adaptive Numerical Environment for Scientific Simulation) [90, 56].

This section is organized into two parts. First we give a brief overview of MADNESS, its k-d tree based data structures, and tree traversal based operators. Second, we present performance comparisons between fused and baseline (unfused) implementations of these operators for benchmark applications in MADNESS.

6.7.1 MADNESS

MADNESS (Multiresolution Adaptive Numerical Environment for Scientific Simulation) [90, 56], which received an R&D 100 Award in 2011, is a general purpose numerical framework for fast computation with guaranteed precision in many dimensions. It is portable from laptops to the largest supercomputers, including the Cray XE and IBM Blue Gene/Q systems, and is open-source under GPL2 with active developers/users at multiple universities and DOE laboratories across the USA, Europe, Japan, and the UK.

MADNESS started as an environment for fast and accurate numerical simulation in chemistry, but rapidly expanded to include applications in nuclear physics [100, 42, 99], boundary value problems [108], solid state physics [128], and atomic and molecular physics in intense laser fields [133].

The modeling capabilities of MADNESS allow science applications to be composed in terms of functions and operators rather than coefficients, grids, or matrix elements.
Spatial Trees in MADNESS

In MADNESS, a mathematical function in space is numerically represented using k-d trees, or spatial trees. The spatial tree stores the coefficient required to represent the function at different region of space. For example, a binary tree is a spatial tree in one dimensional space. Fig. 6.1, shows an example of a numerical function in one-dimension, and the spatial tree representing it.

Coefficients may be stored at the internal nodes of the spatial tree, or only at the leaf nodes based on the representation used. There are two different representation used in MADNESS, 1) reconstructed form or 2) compressed form. In reconstructed form, coefficients are stored only at the leaf nodes of the tree, and represent the scaling function coefficients (scaling function is a particular kind of basis set). In compressed form, coefficients are stored at both interior and leaf nodes. The most common operations in MADNESS are compression, reconstruction, addition, multiplication, inner product, differentiation, and convolution.

Some operators in MADNESS

Listing 6.11: Add Operation for 1D function in MADNESS

```c
ADD(Label, l, ST f1, ST f2, ST o1){
    if(f1[l].is_leaf && f2[l].is_leaf)
        o1[l].coeff = f1[l].coeff + f2[l].coeff;
    if(f1[l].ancestor.is_leaf && f2[l].is_leaf)
        o1[l].coeff = f1[l].getCoeff(l.ancestor) + f2[l].coeff;
    if(f1[l].is_leaf && f2[l].ancestor.is_leaf)
        o1[l].coeff = f1[l].Coeff + f2[l].getCoeff(l.ancestor);
    if(f1[l].is_interior || f2[l].is_interior){
        ADD(l.child0,f1,f2,o1);
        ADD(l.child1,f1,f2,o1);
    }
}
```

To facilitate understanding of the experiments presented here, we focus on addition and multiplication operators computed in reconstructed form. Both of these are binary operators that take two functions as input, and produces a single output.
function corresponding to the sum or the product of the input functions. Lst. 6.11 shows pseudo-code for an add operation. To apply the node operation at a label \( l \), both coefficients from \( f_1 \) and \( f_2 \) have to be available at \( l \). If function \( f_1 \) has a node refined to level \( s \), while the corresponding region in space is only refined to level \( s - 1 \) in \( f_2 \), then \( f_2 \) must be further refined to level \( s \) before the coefficients can be added or multiplied. This is shown in Lst. 6.11 using the `getCoeff` method, which refines the coefficient at a lower level to obtain coefficient at a higher level. The multiplication operator works in a similar fashion.

Add and Multiply operators access their operands using either `SNR` or `AR`, which can produce either \( \psi \) or \( \alpha \) dependences. Thus, a valid fusion schedule can be generated for any sequence of operations involving just Add and Multiply based on Theorem. 6.5.1.

### 6.7.2 Experiments

**Benchmarks:** We show results on two benchmark applications: i) *Sum* and ii) *OuterSum*. The first application computes the sum of a vector of functions. The second application computes the outer-product between vector of functions and itself, then adds all the resulting functions together after scaling them with unique scalars. Both benchmarks were chosen to represent large chain of operations in real applications. The computation can be written at a high level using primitive operators, add and multiply, as follows:

\[
\text{Sum} \; : \; \text{result} = \sum_{i}^{n} F[i] \tag{6.14}
\]

\[
\text{OuterSum} \; : \; \text{result} = \sum_{i,j}^{n} c_{ij} F[i] \times F[j] \tag{6.15}
\]
The size of the vector of functions $F$ is given by $n$, and $c_{ij}$ is a scalar value corresponding to indices $i$ and $j$. For the scalability runs presented below, we set $n$ to 50 and 16 for Sum and OuterSum benchmarks, respectively. These number allowed for the largest run possible.

**Platform:** The experiments were run on a Intel(R) Xeon(R) CPU E5-2687W with 16 cores (8 cores per socket and 2 socket), and shared L3 cache size of 20MB. MADNESS runtime was used to parallelize our implementation. The MADNESS runtime maps each recursive call in a tree traversal to a different core.

### 6.7.3 Performance Comparison

**Performance Improvement:** For the two benchmarks, Fig. 6.3a shows the performance improvement of fused tree traversal over the baseline for varying number of functions on 16 cores.

**Scalability:** Fig. 6.3b shows the scalability of the fused traversal in comparison to the baseline on up to 16 cores.

**Discussion:** We identified three major factors involved in the improved speedup and scalability:

i) The primary source of speed up is a significant reduction in memory I/O. For OuterSum, the fused traversal has an I/O of $\Omega(\sum_{i=1}^{n} |F[i]| + |result|)$, corresponding to the true input and output of the benchmark. The I/O of the fused schedule increases linearly with the number of functions. On the other hand, the baseline has an I/O of $\Omega(\sum_{i=1}^{n} |F[i]| \times n + |result| + \sum_{j=1}^{n^2} 2|I[j]|)$, where $I[j]$ is the $j^{th}$ intermediate function produced. The baseline produces $n^2$, intermediate functions corresponding to each of the element in the outer-product. Thus, the I/O of the OuterProduct for
the baseline increases quadratically as the number of functions are increased. If we assume that the size of each function is approximately equal to \(|f|\), then the total I/O for fused schedule is \(\Omega(n|f|)\), while it is \(\Omega(n^2|f|)\) for the baseline. Thus, we see a strong linear speed up for this benchmark. We believe that the sharp increase in speed up at \(n = 20\), is a result of I/O bottleneck shifting from L3 cache to memory. For \(sum\), the fused traversal has an I/O of \(O(\sum_i |F[i]| + |result|)\), while the baseline has an I/O of \(O(\sum_{i=1}^n |F[i]| + |result| + \sum_{i=j}^{n-1} 2|I[j]|)\). While, the fused schedule has a lower I/O than the baseline, they are still of the same order, and so we see a slower increase in speed up for \(Sum\) than \(OuterSum\).

ii) The fused traversal reduces the number of synchronizations across threads. In the original version, there is a global synchronization at the end of each binary operation, but in the fused version a synchronization is only needed at the end of the entire sequence of operations.

iii) The fused version also increases the granularity of task at each node, allowing for each core to stay busy. Improvement of granularity and the reduction in synchronization, along with improved I/O results in improved scalability as well as performance.

**Summary:** Our experimental results show that fused tree traversal schedules for sequence of primitive operators can be an order of magnitude faster than its unfused execution. The improvement in performance is a result of improved data locality, reduction in the number of global fences, and increase in granularity of computation. The ability to eliminate multiple performance limiting factors shows the potential of fused tree traversals to greatly enhance performance of tree based applications such as MADNESS.
6.8 Related Work

There has been significant research on determining the existence of dependences, characterizing the nature of the dependences, and transforming programs while preserving the dependences. Starting with programs that operate on dense multidimensional arrays and consist of loops with affine loop bounds and array reference expressions, a vast body of work looked at the problem of legality, applicability and profitability of fusion. Earlier works on loop fusion such as [47] focus on the selection of fusion structures with the goal of optimizing for array contraction via a maxflow-mincut algorithm. [92] develop a loop transformation framework and cost model that exploits group reuse by estimating the number of accessed cache lines, while [33] performs an in-depth theoretical study on the complexity of loop fusion considering different objectives (e.g., number of partitions) and enabling transformations (e.g., loop shifting). More recent works have exploited the power of affine transformations on polyhedral models to design advanced fusion schemes. Bondhugula et al. proposed a polyhedral loop fusion and code motion model that optimizes for locality, stream prefetching and parallelism [15], while Pouchet et al. provided a framework to build the set of all and only legal fusion / distribution choices, under arbitrary affine iteration reordering [103]. However, all these techniques are limited to static/affine dataflow and cannot handle tree-based data structures like the one addressed in this chapter.

The elimination of intermediate trees (or lists) and the optimization of recursive traversals has been a central topic in functional programming, for the purpose of converting at compile-time inefficient recursive programs into more efficient ones. Deforestation, such as with treeless form [134], foldr/build [51], unbuild/unfoldr [127], or
stream fusion [31] has been proposed. To preserve functional programming paradigms while avoiding their inefficient execution, precise rewrite rules are applied to convert the program and such techniques have been implemented in the GHC Haskell compiler [72] for instance. However, our work is aimed to a different problem: finding a fusion of operator DAGs on (spatial) tree traversals that purportedly optimizes for data locality. In particular, we can express the set of all and only fusible operators (Thm. 6.5.1) for arbitrary OpDAGs, and search for a globally-optimal fusion (under the restrictions of heuristic in Lst. 6.10) optimizing data locality. This contrasts with deforestation works which, to the best of our knowledge, are not considering all possible (valid) sequences of rewrite rules that can lead to fusion of traversals, in turn discarding possible solutions. In addition, our work deals with imperative programs, such as C/C++, by exposing the necessary and sufficient semantics rules (e.g., dependences) to enable tree traversal fusion.

Ghiya et al. [50] presented analyses to check that function calls access disjoint sub-pieces of tree-like data structures. Our approach to encoding accesses as regular expressions is similar to the approach by Larus and Hilfinger [81], who employ access path expressions to encode data access to check for existence of dependences. Access paths are also employed by Hummel et al. [62]. Rugina et al. [110, 111] presented static analyses techniques to analyze pointer and array index references to parallelize recursive programs. Rinard and Diniz [109] developed commutativity analysis to parallelize programs that manipulate pointer-based data structures despite the presence of dependences. These papers focus on checking for the existence of dependences, with focus on parallelization, but not on a compact representation of their structure to enable subsequent data locality optimizations.
Burstall and Darlington presented a transformation system for recursive programs [17]. Nandivada et al. [94] presented transformations to optimize task parallel programs. Neither approach considers fusion or takes data locality into account. Jo and Kulkarni [71] exploit access information to optimize data locality across concurrent pre-order tree traversals. Meyerovich et al. [93] consider the synthesis of parallel fused tree-traversal schedules for attribute grammars, which exhibit simpler access patterns.

Closest to our work, Weijiang et al. [137] presented a hybrid compile-time/runtime approach to analyze recursive pre-order tree traversals to perform locality- and parallelism-enhancing transformations. Our work differ from this approach in three ways: (i) we consider pre- and post-order traversals, (ii) we employ a compact compile-time characterization of dependences appropriate for loop fusion, and (iii) we present a solution to the problem of fusing such tree traversals.

6.9 Conclusion

In this chapter, we have developed an approach to characterize data dependences for a class of recursive programs that traverse k-ary tree data structures, along with the establishment of necessary and sufficient conditions for the validity of fusion across a sequence of tree traversal operators. We described the FuseT source-to-source compiler framework for automatic transformation of a sequence of primitive operators on k-ary trees into a sequence of composite fused tree-traversal operators. Experimental evaluation on MADNESS benchmarks demonstrated significant performance improvement over the current unfused implementation in the production MADNESS system.
Acknowledgments

This work was supported in part by the U.S. National Science Foundation through award ACI-1440749, and the U.S. Department of Energy’s (DOE) Office of Science, Office of Advanced Scientific Computing Research, under award DE-SC0008844 and DOE Early Career awards 63823. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.
7.1 Introduction

As the scientific community grapples with the “energy first” paradigm [30] that is widely expected to dominate the design of future computing systems, there is at least one point on which substantial consensus has already emerged: “There is no known alternative to parallel systems for sustaining growth in computing performance; however, no compelling programming paradigms for general parallel systems have yet emerged.” [45]. However, the number of applications and disciplines benefiting from parallel computers is being severely limited by the expertise and effort required to create parallel applications.

A case in point that illustrates the massive effort required to develop high-end applications is the MADNESS (Multiresolution Adaptive Numerical Environment for Scientific Simulation) system [90]. MADNESS provides a general purpose numerical environment for a very broad set of applications, roughly characterized by the solution of differential and integral equations in multiple dimensions using adaptive meshes and fast solvers on trees. A prototype code for the then novel numerical approach [7, 55] was developed in about 6 person-months in Python, but the production C++ code to enable real science advances took over 6 person-years of effort. The core data
structures in MADNESS are irregular k-d trees, used for compact representation of numerical functions over multidimensional spaces. The key challenges of expressing novel and still evolving numerical algorithms and representations, while achieving good scalability for the compute functions traversing irregular spatial trees, required the development of an elaborate runtime environment from the ground up. The runtime environment includes an active messaging layer, distributed data management customized for spatial trees, and extensive asynchronous execution and communication.

MADNESS has current applications in nuclear physics [2, 100, 42, 99], boundary value problems [108], chemistry [55, 56, 145, 115], solid state physics [128], and atomic and molecular physics in intense laser fields [133]. However, the current implementation of MADNESS faces significant scalability challenges due to high overheads for dynamic synchronization and data movement.

In this work, we present a layered domain-specific compiler for MADNESS, framed as an embedded DSL system. The C++ MADNESS code can run unmodified, or through the new framework that analyzes and optimizes operator sequences on k-d trees. The new optimizations address key performance bottlenecks of massively distributed applications in MADNESS, and include the fusion of k-d tree traversals and elimination of temporaries to eliminate communications, improve data locality and reduce storage, and domain-specific loop transformations, such as in the MADNESS outer-product of inner-products, an expensive and widely used operation. We make the following contributions:
We introduce a MADNESS-specific C++ embedded DSL system to describe a) operator specifications including neighbor dependences, and b) sequences of operators that can be analyzed and optimized.

We develop automatic analyses and optimizations of MADNESS operators to automatically enable semantically correct fusion of operator traversals, elimination of temporaries, and loop optimizations, and provide runtime support for these optimizations.

We demonstrate significant performance improvements over distributed MADNESS applications, on clusters of up to 2160 cores, with time improvements of up to 4x.

The rest of the chapter is organized as follows. Sec. 7.2 presents background on MADNESS and k-d tree optimizations, and an overview of the approach. Sec. 7.3 introduces the MADNESS embedded DSL, and Sec. 7.4 describes the optimizing compiler. Experimental results are presented in Sec. 7.5, followed by related work in Sec. 7.6.

7.2 Background and Overview

7.2.1 MADNESS Framework

MADNESS is a high-level software environment for the solution of integral and differential equations. It uses k-d trees for representation of functions over k-dimensional space. Interpolation polynomial coefficients are stored at nodes (leaves) of an adaptively refined k-d tree and can be used to compute the numerical value of the function at any spatial location of interest. Fig. 7.1 shows an example of a 1-d function and an adaptively refined binary tree representing the function to a desired accuracy. The
function is approximated by different polynomials in different regions. The size of regions captured by the leaves of the binary tree can be seen to vary – spatial regions with more rapid variation of the function are more deeply refined than smoother regions. In this example, the function in the spatial range \([1/2:1]\) is represented by a single polynomial, while the function in the x-range \([0:1/2]\) is split into five regions: \([0:1/8]\), \([1/8:1/4]\), \([1/4:5/16]\), \([5/16:3/8]\), and \([3/8:1/2]\), based on how rapidly the function varies in the different regions.

Nodes in a k-d tree can be labeled in a systematic manner, as illustrated in Fig. 7.1(b). The root node is labeled 1, and if a node with label \(l\) has children, the left child has label \(2*l\) and the right child has label \(2*l+1\). The labeling convention extends to k-d trees that are used to represent k-dimensional spatial functions by using a k-tuple for the label. The root node has a label of \((1,1,...,1)\), and each refined node with label \((l_d,l_{d-1},..,l_i,.,l_1)\) has \(2^k\) children, with all possible labels derived from the refined node, where label \(l_i\) is replaced by either label \(2*l_i\) or label \(2*l_i+1\).

MADNESS operators are specified as recursive functions operating on k-d trees representing input operands (fields). Each operator takes as input one or more fields (k-d trees) and produces an output field (a k-d tree). Each node of the output tree is computed using values from “corresponding” nodes on the input k-d trees. If the input k-d trees have identical structures, the output k-d tree will also have identical structure to the input trees, and the value stored at a node with label \(l\) in the result tree is computed using the values at the nodes with the same label \(l\) in the input trees. When the input trees do not have matching structures, as is typically the case, the output tree is created with sub-domains that are intersections or unions of sub-domains in the input trees, depending on the operator.
In the current MADNESS implementation, the recursive functions corresponding to the operators are executed in a nested fork-join (or recursive parallel) fashion, corresponding to parallel pre-order or post-order traversal of the trees.

7.2.2 FuseT Compiler Framework

The nested fork-join model leads to synchronizations between operators that operate on shared trees (often through a producer-consumer relationship). While explicit synchronization for dependences on individual nodes can avoid a global synchronization, prior work shows that the nested structure can be exploited to perform data locality optimization. Specifically, in our prior work [106], it was observed that recursive traversals of k-d trees can be fused to improve cache locality, with demonstration
in a shared-memory parallel implementation, in the context of kernels from MADNESS. That fusion framework, FuseT, is summarized below.

In FuseT, an operator specification consists of two parts: i) a compute method that specifies how an output field is computed at a given spatial sub-domain, referenced by using a node label in the k-d tree, and ii) data access patterns that specify the relative positional relationship of the node labels of needed components from input fields to the node label of a component of the output field.

In a sequence of operators with a producer-consumer relation, the data access pattern determines the type of dependence between the operators. Four types of dependences are modeled in FuseT, corresponding to four different data access patterns: i) ancestor dependence ($\alpha$), ii) descendent dependence ($\delta$), iii) self-node dependence ($\psi$), and iv) some-sibling dependence ($\sigma$).

Let $pOp$ and $cOp$ be producer and consumer operators, respectively. An ancestor dependence ($cOp \leftarrow^\alpha pOp$) exists between them if the computation of the output field of $cOp$ at node label $l$ (represented as $cOp[l]$) accesses the output field of $pOp$ at an ancestor node of node $l$ (represented as $pOp[l.\text{ancestor}]$). Similarly, $cOp \leftarrow^\delta pOp$, or $cOp \leftarrow^\psi pOp$ if the computation of $cOp[l]$ accesses $pOp[l.\text{descendent}]$, or $pOp[l]$, respectively. Finally, $cOp \leftarrow^\sigma pOp$, if $cOp[l]$ accesses $pOp[l']$, where $l' \notin \{l, l.\text{ancestor}, l.\text{descendent}\}$.

The FuseT compiler performs dependence analysis to determine the fusibility of a sequence of operators. A sequence of operators is represented as an Operator Directed Acyclic Graph (OpDAG). The vertices of the graph represent operations, and the edges represent dependences. It was previously shown [106] that an OpDAG is fusible iff the sequence of dependences in all paths between any two operators in the
OpDAG can be recognized by the regular expression $R_{\text{valid}} : (\delta^* \psi^*)^*(\alpha^* \psi^*)^*$. Under the FuseT framework, all $\alpha$ dependences must precede all $\delta$ dependences, and there can be no $\sigma$ dependences.

### 7.2.3 MADNESS Runtime

The MADNESS runtime consists of i) a task scheduler for scheduling parallel tasks on a distributed system, and ii) support for dynamic satisfaction of data dependences between tasks. A task that is ready to run can be pushed to the task queue with its parameters and a destination processor ID. This step is non-blocking, and the program control returns immediately. The runtime serializes the parameters and sends the task to the destination processor, where the parameters are deserialized and the task is executed.

A task that is dependent on parameters that will only be produced at a later time can be submitted by passing *futures* as parameters. If a future is passed as a parameter to a task, that task is only executed after the future is assigned. Tasks can also return futures. By passing the future returned by one task as a parameter to another task, data dependences between the two tasks are satisfied.

In this paper, we show that the specific properties of MADNESS operators can be exploited to perform the following optimizations:

1. Careful interleaving of MADNESS operators to partially compute, use, and delete intermediate trees, reducing the overall memory footprint for MADNESS programs,

2. Reduce the overall operation count by dynamically reordering associative operators.
We demonstrate these optimizations using an implementation in the production MADNESS framework.

7.2.4 MADNESS Challenges and Solution Overview

While the MADNESS framework and the FuseT compiler simplify implementation of complex operators, there are several limitations. The computation to be performed is structured as a sequence of MADNESS operators, each with a tree traversal. While the FuseT compiler identifies several opportunities to fuse the operator traversals, functions with spatial dependences—dependences on spatial neighbors of a tree node—are not handled by FuseT. These represent widely used and computationally expensive operators in MADNESS.

MADNESS operators compute on functions that are created, produced, and destroyed as a unit. This can lead to significant memory requirements to store complete functions until they are used. FuseT's operator specification and transformations rely on information about the definition and first use of individual tree nodes to perform fusion. They are insufficient to identify their last use, to reclaim allocated memory.

Finally, while the transformation implemented by FuseT can reduce reuse distance and increase cache locality, it does not recognize opportunities to increase compute intensity (e.g., computing a sequence of 2-d loop nests as a 3-d loop nest instead). In this chapter, we address these challenges by making the following contributions:

- We extend the operator specification to describe neighbor dependences, with analysis of the necessary and sufficient conditions to fuse operators with such dependences.
• We develop a fused tree traversal template that maps the structure of the MADNESS functions into method invocation parameters, allowing eager memory reclamation for intermediate functions.

• We create a domain-specific optimization framework that takes as input a sequence of operators (tree traversals) on MADNESS spatial trees, and determines the fused schedule together with the elimination of intermediates through the analysis of their last use.

• We develop a permutation transformation for MADNESS outer-product of inner-products, an expensive and widely used operation. This is combined with an approach to recognize and transform a sequence of operations at a particular label into a higher-dimensional analogue that achieves better compute intensity and improved data locality.

7.3 MADNESS Embedded DSL

The MADNESS embedded DSL infrastructure, depicted in Fig. 7.2 comprises of three interconnected components: (1) The Operator specification, a C++ interface to implement MADNESS operators, where data access information is specified as part of the implementation, to enable dependence analysis and transformation; (2) The Traversal runtime, a scalable runtime system to perform any MADNESS tree traversal computation, structured to affect the execution schedule based on the input from the DSL optimizer; (3) The X-Compiler, an embedded DSL optimizer that takes a sequence of operators and their definitions as input, and generates an execution schedule via optimizations including generalized operator traversal fusion, intermediate elimination, and loop permutation.
A MADNESS application only needs to be slightly modified to use the new operator specification. In particular, the specification does not impact the computation being performed (the kernel to operate on two tree nodes), but only the structure of the implementation (i.e., the functions used to implement them), and data access annotations to enable further optimizations. The transformations presented in this chapter take an input sequence of operators and execute them as a collection of fused operators. The fused operator execution respects dependences, optimizes data reuse and cache locality, and minimizes total memory usage by freeing up memory as soon as all of its uses are complete.
7.3.1 Operator Specification

Implementing a MADNESS operator using the framework involves two parts: (i) implementing the operator by inheriting from a generic operator interface and (ii) specifying the data access characteristics of the operator.

MADNESS operators are structured as either pre-order (compute a node before any of its children) or post-order (compute a node after all its children) tree traversals. The order depends on the dependences. For example, if computing a node requires the result at its parent node, a pre-order traversal is used. If all nodes in a tree can be computed independently, we allow them to be specified in either traversal order.

Listing 7.1: Virtual methods of Operator base class

```cpp
virtual Para compute (KeyT key, const Para &s)=0;
virtual Future<Para> computeFuture(KeyT key,
const Para &s)=0;
virtual bool isDone(KeyT key) const = 0;
virtual bool isPre() const = 0;
virtual void reduce(World& world)= 0;
virtual int accessType()=0;
virtual void erase(KeyT key)= 0;
```

Lst. 7.1 shows the interface (as an abstract base class) to be implemented by each MADNESS operator. Para is a container type to hold a collection of types that can be passed through the recursive function calls. The compute method performs the operator's computation on the tree label key. The compute method is designed to be completely local and does not invoke any remote communication. The method can directly access the node with label key in any tree that is an input to or output of the operator. MADNESS's data distribution places all nodes with the same key (in any tree) in the same processor. It also has access to any other node in these trees—ancestors, descendants, spatial neighbors, etc.—through appropriate use of the Para parameter. The compute method returns a collection of nodes in different trees. Note that this is a generic interface that enables the optimizations presented in this
chapter. In a typical implementation by the user, the `compute` method returns a single node corresponding to the output of the operator for the given node with label key.

If the output field at a particular key depends on input fields at a different key (possibly stored in a different processor), or on keys that will be produced at a later time, executing such operators can require communication. Such operators are implemented using the `computeFuture` method. This method can initiate remote communication to acquire input fields at remote keys or to push an input field at a local key to a remote processor. The method returns a future to be processed when needed operands are available in local memory.

Depending on the desired accuracy and the availability of operands, a given operator might not require any computation beyond a particular node in the tree. For example, when multiplying corresponding nodes in two trees, if one operand tree has no node for a given key, indicating it is zero, the multiplication need not be performed. The `isDone` method in an operator includes the logic to determine the condition for termination of the recursive traversal of a subtree. The traversal pre- or post-order is specified using the `isPre` method.

**Listing 7.2: Data access types**

```java
enum AccessType {
    // bits or-ed to produce an access type specification
    self = 0b1, // same node
    alpha = 0b10, // ancestor
    delta = 0b100, // descendant
    NA = 0b1000, // spatial sibling
    ANA = 0b10000, // ancestor’s spatial sibling
    DNA = 0b100000 // descendant’s spatial sibling
};
```

Some MADNESS operators require a global reduction to be performed on a tree once it is computed. This operation is specified using the `reduce` method.
This operator interface can be used to implement all MADNESS operators, including reconstruct, compress, project, inner-product, derivative, truncate, add, multiply, subtract, divide, norm, trace, and scale with constant.

**Data access specification** To enable automated transformation of tree-traversal operators, operators must provide information on how each operator accesses data from its input fields. As specified in Sec. 7.2, FuseT considers three data access types: ancestor-access \((\alpha)\), descendent-access \((\delta)\), and self-access \((\psi)\). We introduce three additional data access types corresponding to sibling data access \((\sigma)\):

- **Neighbor Access (NA)**: \(l'.depth == l\).depth. With NA, node label \(l\) accesses node label \(l'\) at the same depth as \(l\).

- **Ancestor’s Neighbor Access (ANA)**: \(l' = l.ancestor.neighbor\). With ANA, node label \(l\) accesses node label \(l'\) that is a neighbor to an ancestor of node label \(l\).

- **Descendent’s Neighbor Access (DNA)**: \(l' = l.descendent.neighbor\). With DNA, node label \(l\) accesses node label \(l'\) that is a neighbor to a descendent of node label \(l\).

This access type specification is provided by implementing the `AccessType` method. As we will show, these three data access types enable the fusion of operator sequences with \(\sigma\) dependences.
Listing 7.3: Derivative operator

SpatialTree input, output  //input/output for this operator
Future<Para> computeFuture(KeyT key, const Para &s) {
    //left, right: coeff from left and right node labels
    Future<Coeff> left, right, center;
    if(!Para[LEFT].empty()) left = Para[LEFT];
    else
        //remoteFetch returns coeff at key.left possibly
        //from another processor. sibling access
        left = task(owner(key),remoteFetch,key.left);
    //similar code as above for Para[RIGHT]
    if(!Para[CENTER].empty()) center = Para[CENTER];
    else center = input.getCoeff(key);  //same node access
    return task(owner(key),compDeriv,left,center,right);
}
Future<Para> compDeriv(key, left, right, center) {
    output.isLeaf(key) = false;
    if(!(left.empty(), center.empty(), right.empty()))
        output.replace(key,derivative(left,center,right));
    output.isLeaf(key) = true;
    //allows ancestor and ancestor-sibling access
    return /*Para object with left, right, and center*/
}
bool isDone(const KeyT& key){
    return output.isLeaf(key);}
bool isPre(){return true;}
int accessType() { return same|alpha|NA|ANA;}

Illustration   Lst. 7.3 shows the operator specification for a derivative operator. A derivative operator is computed in reconstructed form, where coefficients are only stored at leaf nodes. Computing the derivative at a node label requires coefficients at that node label along with coefficients at labels to its left and right. If one of these coefficients does not exist, then no computation is necessary at the node label. In addition, if coefficients are found at a node label, then it implies that the node label is a leaf node, and no coefficients exist at its descendants. However, coefficients at children node labels can be generated from coefficients at their parent node label. Therefore, if coefficients are found, they are passed to its children as parameters.

Note that the derivative operator can require access to nodes in the input tree in four different ways: to compute derivative at node label l, it can require input coefficients at node label l, l’s ancestor, l’s siblings, or l’s ancestor’s siblings.
7.3.2 Traversal Runtime

Listing 7.4: Single operator traversal

```cpp
def treeTraversal(KeyT key, Para s):
    rV = op->compute(key, s);
    rV = op->computeFuture(key, s);
    return task(owner(key), &continueTraversal, key, rV);
    rV = op->compute(key, s);
    Para preR = rV.get();
    if (!op->isDone(key))
        Future<Para> postR[];
        for (KeyT k : key.child)
            postR[k] = task(owner(k), treeTraversal, k, preR[k]);
    if (!op->isPre())
        return task(owner(key), &postCompute, key, postR);
    return Para();
}
```

The traversal runtime is constructed using MADNESS tasks and futures. It provides support for execution of individual operators using a single operator traversal, or execution of a sequence of fusible operators using a fused operator traversal.

Single Operator Traversal

Individual operators are executed using the single operator traversal shown below in Lst. 7.4. The execution begins by invoking the `traverseTree` method as a MADNESS task at the root key. If the operator is a pre-traversal operator, then its `compute` (or `computeFuture`) method is executed immediately, and the return object acts as the parameter to the recursive calls to `traverseTree`.

If the operator returns a future, then `traverseTree` returns after pushing the `continueTraversal` method to the task queue. This method simply resumes
traverseTree from the line marked as the continuation point. As continueTraversal is only executed after the future returned by computeFuture is assigned, continueTraversal provides a mechanism for pausing the execution of traverseTree until the pre-computation is completed.

**Listing 7.5:** Fused Operators Traversal

```cpp
//mapP : map<OpID, Para>
Future<mapP> fusedTraversal(keyT key, fusedInfo fI, mapP s)
{
    fusedInfo newfI;
    mapP prePara[NumOfChildren];
    for(OpID id : fI.preCompute)
        if(opSeq[id]->computeFuture)
            Future<Para> preR = opSeq[id]->computeFuture(key, s[id]);
            State st = SaveState(key, fI, s, newfI, id);
            return task(owner(key), &continueFTraversal, state, preR);
    future = computeFuture(key, s);
    preR = opSeq[id]->compute(key, s[id]);
    //continuation point
    if(!opSeq[id]->isDone(key))
        newfI.preCompute.push_back(id);
    for(keyT k : key.child)
        prePara[k][id] = preR[k];
    newfI.postCompute = findPostComputeAtChildren(key);
    for(OpID id : fI.preCompute)
        if(opSeq[id]->preInterim) opSeq[id]->erase(key);
    if(!newfI.preCompute.empty() || !newfI.postCompute.empty())
        Future<mapP> pR[NumOfChildren];
        for(keyT k : key.child)
            pR[k] = task(owner(k), &fusedTraversal, k, newfI, prePara[k][id]);
        return task(owner(key), &fPostCompute, fI, pR);
}
Future<mapP> fPostCompute(keyT key, fusedInfo fI, mapP pR[])
{
    mapP s = getPara(postR);
    mapP postMap;
    for(OpID id : fI.postCompute)
        if(opSeq[id]->computeFuture)
            Future<Para> opR = opSeq[id]->computeFuture(key, s[id]);
            State st = SaveState(key, postMap, fI, id);
            //Future dependent task that resumes from post continuation point
            return task(owner(key), &continuePCompute, state, opR);
    for(keyT k : key.child)
        postMap[id] = opR;
    return Future<mapP>(postMap);
}
Future<mapP> continueFTraversal(State s, Para preR)
{
    //continue treeTraversal from continuation Point
    ...;
    Future<mapP> continuePCompute(State s, Para postR)
        //continue treeTraversal from post continuation point
        ...
}
```
For a post-compute operator, a similar mechanism allows the `postCompute` method to be invoked at a key once the execution of `traverseTree` at its children are completed. The parameter to the `postCompute` method is the composite of return objects from execution of the `compute` methods at its children nodes.

**Fused Operator Traversal**

A sequence of fusible operators can be executed with a single tree traversal using the fused traversal algorithm shown in Lst. 7.5. It has three major features: i) it supports parameter passing and return, ii) it can eliminate intermediates, and iii) it can execute fused operators with $\sigma$ dependences.

**Parameter Passing** In a similar manner to single operator traversal, the fused traversal runtime supports parameter passing. Return objects from the pre-traversal operators are passed as parameters to the pre-traversal operators at the children nodes. Similarly, return objects from the post-traversal operators are passed as parameters to post-traversal operators at the parent key.

**Intermediate Elimination** The fused traversal runtime supports intermediate elimination. As we will see in Sec. 7.4.2, parameter passing allows MADNESS developers to write single access operators that access input fields exactly once. If an operator is single access, the X-Compiler can determine when it can be removed. The `fusedTraversal` removes output fields of operators marked as `preInterm` at the end of the pre-traversal phase, and it eliminates output fields of operators marked as `postInterm` at the end of the post-traversal phase.
**Fusing Sibling Dependence**  Given a producer and consumer operator pair, a $\sigma$ dependence exists between them if the output field of the consumer operator at a key depends on the output field of the producer operator at a sibling key. Notice that fusedTraversal executes in parallel at sibling keys. To satisfy the sibling dependence ($\sigma$), the consumer operator at a key must wait until the producer operator completes its computation at the sibling key. This is achieved with the fused operator runtime by suspending fusedTraversal until the consumer operator completes its computation.

Operators that require any form of waiting are implemented using computeFuture. The fused operator traversal can execute such an operator and suspend itself until the operator completes. By resuming after the operator completes, any dependence between this operator and any other remaining operator in the fused sequence is also satisfied. The suspension and resumption mechanism are implemented using continueFTraversal and continuePCompute in Lst. 7.5.

### 7.4 Domain Specific Fusion Compiler

#### 7.4.1 Fusing operators with near neighbor dependence

In this section, we present the fusion compiler based on FuseT [106], extended to support fusion of operator sequences with some sibling dependences ($\sigma$). To do so, we first formalize the ordering of operators in a fused traversal, presented in Sec. 7.3.2. Next, we present a characterization of different types of sibling dependences, and establish the necessary and sufficient conditions for an operator sequence with such dependences to be fusible. Finally, we show how to create a fused schedule that satisfies these dependences.
Figure 7.3: The figure shows the ordering between pre- and post-traversal operators enforced by fused traversal and sibling dependences. It also shows when a sequence of operators with sibling dependences can be fused, and when they cannot.

A tree traversal that fuses a collection of operators needs to satisfy the following conditions, illustrated in Figure 7.3:

- A pre-traversal operator at node label $l$ always executes before a pre-traversal or post-traversal operator at $l$’s descendent node (shown in Fig. 7.3(a)).

- A post-traversal operator at node label $l$ always executes after post-traversal operators at its descendent node labels (shown in Fig. 7.3(b)).

Additionally, an operator at a node label $l$ can require another operator at a different node label $l'$ to execute before it, as long as this ordering does not violate
the two constraints above. Otherwise, a cyclic dependence could be created, leading to deadlock - such operators cannot be fused together.

**Types of some-sibling \( \sigma \) dependences**  In addition to same label, descendent, and ancestor dependences, operators on spatial trees might operate on their spatial neighborhood. We refer to such dependences as sibling (\( \sigma \)) dependences, and classify those dependences into three types based on the three some-sibling data access types:

- **Neighbor Dependence** (ND or \( \lambda \))
- **Ancestor’s Neighbor Dependence** (AND or \( \mu \))
- **Descendent’s Neighbor Dependence** (DND or \( \nu \))

For example, \( cOp \leftarrow^\lambda pOp \) implies that computation of \( cOp[l] \) accesses \( pOp[l'] \), where \( l \) and \( l' \) are spatial neighbors. Fig. 7.3(c, d) shows \( \mu \) and \( \nu \) dependences.

**Necessary and Sufficient Conditions for Fusibility**

Fusing operators by exploiting their sibling dependences involves determining the conditions under which these operators can be fused. Once the fusibility conditions are identified, the FuseT framework can be used to construct the fused operator schedule. Therefore, we establish the conditions under which an operator sequence with sibling dependences is fusible.

A sequence of operators is fusible if a partial ordering of operators exists that satisfies both the ordering constraints of the fused traversal, and the dependences between the operators. The necessary and sufficient conditions for fusibility are given by the following theorem:
**Theorem 7.4.1.** A sequence of operators is fusible iff all dependence chains in the sequence can be recognized using the following regular expression

\[ R_{\text{valid}} : (\nu^*\psi^*)^*(\delta^*\psi^*)^*(\nu^*\psi^*)^*(\mu^*\psi^*)^*(\alpha^*\psi^*)^*(\mu^*\psi^*)^* \]

Equivalently, a sequence of operators is fusible iff

- **Condition 1:** All \( \alpha \) dependences precede all \( \delta \) dependences \[106\]
- **Condition 2:** All \( \alpha \) dependences precede all \( \nu \) dependences
  
  \( \text{(cannot have } \leftarrow^\alpha \leftarrow^\psi ... \leftarrow cOp \leftarrow^\nu \text{)} \)

- **Condition 3:** All \( \mu \) dependences precede all \( \delta \) dependences
  
  \( \text{(cannot have } \leftarrow^\mu pOp \leftarrow^\psi ... \leftarrow^\delta \text{)} \)

- **Condition 4:** All \( \mu \) dependences precede all \( \nu \) dependences
  
  \( \text{(cannot have } \leftarrow^\mu pOp_\mu \leftarrow^\psi^*\delta^* ... cOp_\nu \leftarrow^\nu ...) \)

**Proof of The Necessary Condition**  We refer the reader to Rajbhandari et al. \[106\] for proof of Condition 1. It is shown that violating Condition 1 also requires violating the ordering constraints of the fused schedule.

We use additional results from that work to prove Conditions 2 and 3:

- If an \( \alpha \) dependence follows \( cOp \), then \( cOp \) must be a pre-traversal operator.
  
  Thus, violating Condition 2 makes \( cOp \) a pre-traversal operator.

- If a \( \delta \) dependence exists in the dependence chain that leads to \( pOp \), then \( pOp \) must be a post-traversal operator. Thus, violation of Condition 3 makes \( pOp \) a post-traversal operator.
Proof. We use these results to prove that violating Conditions 2 or 3 creates a cyclic dependence that could lead to a deadlock. Then we prove that violating Condition 4 also leads to deadlock.

a) Condition 2: If the consumer operator is a pre-traversal operator and has \( \nu \) dependence with a producer operator, then it leads to a deadlock. This is shown in Fig. 7.3(g). Violating Condition 2 makes the consumer operator \((cOp)\) a pre-traversal operator, hence causing deadlock.

b) Condition 3: If a consumer operator has \( \mu \) dependence with a post-traversal producer operator, then it leads to a deadlock. This is shown in Fig. 7.3(h). Violating Condition 3, makes the producer operator \((pOp)\) a post-traversal.

c) Condition 4: Violating Condition 4 means that there exists an operator \( pOp_\mu \) that follows an operator \( cOp_\nu \), such that \( pOp_\mu \) is a producer operator in a \( \mu \) dependence, and \( cOp_\nu \) is a consumer operator in a \( \nu \) dependence. Additionally, all dependences between these two operators are either \( \psi \) or \( \delta \). This is shown inside parentheses at the end of Condition 4 in Theorem. 7.4.1.

We note that from Fig. 7.3(f,g) \( cOp_\nu \) must be a post-traversal operator to avoid deadlock. This forces \( pOp_\mu \) to also be a post-traversal operator, which leads to a deadlock (Fig. 7.3(h)).

Proof of The Sufficient Condition  If a sequence of operators contains dependence chains that are all recognizable by \( R_{valid} \), then it can be fused. The proof relies on construction of a valid fused schedule for the operator sequence. A valid fused schedule must satisfy all dependences in the dependence chain, and it must avoid
deadlock. We construct such a fused schedule by placing operators either in pre or post Op such that deadlocks are avoided and all dependences are satisfied.

Proof. Let $O_{Seq}$ be a sequence of operators whose dependence sequences are recognizable by $R_{valid}$. A valid fused schedule can be constructed by partitioning $O_{Seq}$ into two sub-sequences, and placing the operators in the first and second sub-sequence in postOp and preOp, respectively. The partitioning is done such that the first and second sub-sequences can be recognized using regular expression $L$ and $R$, respectively, where $R_{valid} = LR$, and

\begin{align}
L &= (\nu^* \psi^*)^* (\delta^* \psi^*)^* (\nu^* \psi^*)^* \\
R &= (\mu^* \psi^*)^* (\alpha^* \psi^*)^* (\mu^* \psi^*)^* 
\end{align}

(7.1) (7.2)

By placing a sub-sequence of operators ($L_{ops}$) recognized by $L$ in postOps in the fused schedule, all the $\delta$ dependences are satisfied (Rajbhandari et al. [106]). Additionally, as all the consumer operators with $\nu$ dependence are also in postOps, the schedule is deadlock free (Fig. 7.3(f)).

Similarly, by placing a sub-sequence of operators ($R_{ops}$) recognized by $R$ in preOps in the fused schedule, all the $\alpha$ dependences are satisfied (Rajbhandari et al. [106]). Additionally, as all the producer operators with $\mu$ dependence are also in preOps, the schedule is deadlock free (Fig. 7.3(e)).

\[\square\]

7.4.2 Elimination of Intermediates

For a fusible sequence of operators, elimination of intermediates means that nodes of the intermediate trees in this sequence are produced and consumed immediately, without the need to store the entire tree. It is difficult to eliminate intermediates
because most MADNESS operators are multi-access operators. In other words, during an operation, the nodes of the operand tree can be accessed numerous times. The number of accesses depends on the type of the operator and the structure of the operands. It is unknown at compile time, and even during runtime; it can only be found by scanning the structure of the operand trees.

For example, consider an operator with ancestor data access type, such as the multiplication operator. With such an operator, nodes of an operand tree at label $l$ can be accessed from multiple node labels $l'|l' \in l.descendant$. Similarly, an operator with descendant access type can access node label $l$ from multiple node labels $l'|l' \in l.ancestor$. It is not possible to know how many ancestors or descendants will access a node without knowing the structure of the operand trees and the specifics of the operation.

To enable intermediate elimination, we take an alternate approach. Instead of accessing a node label multiple times, we use parameter passing and return to implement a multiple-access operator as single-access operators. In this model, an operator computing node label $l$ only accesses the operand tree at node label $l$. However, it can pass data of the operand tree at this node label as parameters to the children node labels. Similarly, it can also return data to the parent node label. By maintaining a local copy of the data and passing it as a parameter or returning it as a return value, a multiple-access operator with ancestor or descendant data access can be implemented as a single-access operator. The local copy of the data can be stored using reference-counting data structures, and deleted after it goes out of scope.

In a fused sequence, if all consumer operators of an intermediate tree are single-access operators, then the intermediate tree will not be accessed at a node label after
the execution of the last consumer operator at that node label. Thus, the fusion compiler can mark when the nodes of the intermediate tree can be deleted. If a pre-traversal operator, whose output is an intermediate tree, is not a producer to any post-traversal operator, then nodes of this intermediate tree can be deleted at the end of the pre-traversal phase (at each node label) during fused traversal. If this operator is a producer to a post-traversal operator, then the nodes of the intermediate tree can only be deleted at the end of post-traversal (at each node label). Similarly, if the operator producing an intermediate tree is a post-traversal operator, then the nodes of the intermediate tree can only be deleted at the end of post-traversal.

During the execution, the fused traversal calls the `erase` method for all the intermediate operators at the end of the pre-traversal computation or post-traversal computation, as marked by the compiler. The `erase` method is implemented by the operator specification to delete the node produced by that operator at a given key.

**Example:** Consider a Producer-Consumer operator pair, where the producer produces an intermediate tree $i_1$. Let the consumer be a multiplication operator that multiplies $i_1$ with some other tree $f_1$. During the fused execution of this operator sequence at node label $l$, $i_1[l]$ is produced by the producer operator. If both $i_1[l]$ and $f_1[l]$ are leaf nodes, then the multiplication operator will complete with no further recursion, and $i_1[l]$ can be deleted. If $i_1[l]$ is a leaf node, but $f_1[l]$ is further refined, then $i_1[l]$ can still be deleted after a local copy is made. The copy can be passed as a parameter by reference to the recursive calls to the children node labels.
7.4.3 Loop permutation enabling affine code optimizations

MADNESS outer-product of inner-products

There are several MADNESS programs that require application of a single operator on a vector of functions. Two such examples are summarized in Lst. 7.6.

Listing 7.6: InnerMatrix and Transpose operators

```cpp
// Inner-matrix Product operator.
double r[N][M]
function f[N], g[M]
generateFunctions(f,g)
for i...
  for j...
    r[i][j] = f[i].inner(g[j])

// Transpose operator.
double c[N][N]
initialize(c)
function f[N], g[N]
generateFunctions(f)
for i...
  for j...
    g[i] += c[i,j]*f[j]
```

As discussed earlier, each operator performs a traversal of the tree that represents the input and output functions of an operator. An inner product requires the traversal of the union of its operand trees, and the computation is performed only on the intersection of the two trees. Elaborating on the previous listing, the actual loop nest for the inner-matrix product operator (Lst. reflst:innerTrans) is shown in Lst. 7.7.

Listing 7.7: Inner Matrix operator

```cpp
for i...
  for j...
    for (nodeLabel n: allNodes in f[i] U g[j])
      if(!f[i][n].empty() && !g[j][n].empty())
        r[i][j]+= f[i][n].transposeConjugate(g[j][n])
```

By leveraging the property that + is an associative operator, and that no dependence exists between loops i and j due to the nature of the MADNESS operator, we can then perform loop permutation in a way that exposes regular loops. This results in Lst. 7.8.
Such a transformation exposes regular loops (via a simple local inspection of the existing nodes, omitted for simplification of presentation) and exposes a matrix-multiplication operation, as $\text{transposeConjugate}$ is simply an inner-product. The resulting loop nest is shown in Lst. 7.9.

Listing 7.8: Inner Matrix operator after permutation

```c
for (nodeLabel n: allNodes in f[i] U g[j])
  // fill-in vectors (n,I) and (n,J):
  // all n,I such that f[I][n] exists, and
  // all n,J such that g[J][n] exists
  // ...
for (nodeLabel n: allNodes in f[i] U g[j])
  for (int i : (n,I))
    for (int j : (n,J))
      r[i][j]+= f[i][n].transposeConjugate(g[j][n])
```

Listing 7.9: Final Inner Matrix operator

```c
// ...
for (nodeLabel n: allNodes in f[i] U g[j])
  // matrix-Multiply:
  for (int i : (n,I))
    for (int j : (n,J))
      for k ...
        r[i][j]+= f[i][n][k]*g[j][n][k]
```

We have explicitly implemented such optimized operations for the outer-product of inner-products in MADNESS, as reported in Sec. 7.5.

Automated transformation opportunities

Due to the semantics of MADNESS tree traversals, exploiting the domain-specific properties of operators allows the compiler to implement both loop fusion and loop permutation in a semantically correct way. The same reasoning can be extended to multiple operator sequences, leading to further loop optimization opportunities.

Another key aspect is that by exploiting the semantics of the operator’s tree traversal to view it as a sparse iteration space, it is possible to form a piecewise-regular iteration space that is a set of dense/affine sub-iteration spaces as shown above when building the I and J subsets. Automated compiler transformations of
7.5 Experimental Results

In this section, we present experimental results for three micro and macro methods implemented with the X-MADNESS DSL, and provide comparisons with the base MADNESS implementation.

7.5.1 Benchmarks

The three benchmarks we consider are MatrixInnerOp, DerivativeOp, and Kinetic Energy Matrix Calculation (KEMC). All three benchmarks are comprised of operations common in MADNESS applications.

MatrixInnerOp (denoted \texttt{m1}) is defined as follows. Given two vectors of functions, this method calculates the inner-product between each function in one vector and each function in the other. It can be written as

\[
R[i, j] = f[i].inner(g[j]) \quad (7.3)
\]

DerivativeSequence (denoted \texttt{m2}) is defined as follows. Given a vector of functions, this method computes the derivative of each vector along \(x\), followed by \(y\), followed by \(z\):

\[
g[i] = \frac{d}{dz} \left( \frac{d}{dy} \left( \frac{df[i]}{dx} \right) \right) \quad (7.4)
\]

KEMC, denoted \texttt{M1}, is a macro benchmark that is the computational bottleneck in Self Consistent Field (SCF) calculations. It takes two vectors of functions, performs
reconstruction on them, followed by computing derivatives along each of the x, y, and z dimensions, followed by a compress operator, and finally a MatrixInner operation described above. It can be written as:

```plaintext
for i
    fr[i] = f[i].reconstruct()
    cdf_x[i] = (df[i]/dx).compress()
    cdf_y[i] = (df[i]/dy).compress()
    cdf_z[i] = (df[i]/dz).compress()
for j
    gr[i] = g[i].reconstruct()
    cdg_x[i] = (dg[i]/dx).compress()
    cdg_y[i] = (dg[i]/dy).compress()
    cdg_z[i] = (dg[i]/dz).compress()
MatrixInnerOp(cdf_x, cdg_x)
MatrixInnerOp(cdf_y, cdg_y)
MatrixInnerOp(cdf_z, cdg_z)
```

The construction of the kinetic-energy matrix is one of three major computational bottlenecks in the solution of the non-linear Kohn-Sham equations, especially for large systems. Our algorithm to iteratively solve the Kohn Sham equations has these steps: 1) relocalization of the orbitals, 2) construction of the potential and kinetic energy matrices; 3) application of the Green functions to compute the residuals; 4) update of the localized orbitals using a non-linear solver; and 5) orthonormalization of the new orbitals using a second-order Newton update. Nominally, several of these steps scale cubically with the system size, but we employ two main techniques to mitigate this. First, the adaptive multiresolution representation in the multiwavelet basis makes it efficient to compute inner products due to the disjoint support of the basis functions and the orthogonality between scales of wavelets. Second, instead of solving for the delocalized eigenstates, we solve for a set of localized states that span the same space. The localization introduces sparsity and the localized states are
in many systems more well defined than the eigenstates, making the iteration more rapidly convergent. Step 1) nominally scales cubically, but has a very small prefactor and after the first iteration re-localizing the already strongly local updated orbitals becomes inexpensive. Allowing for sparsity and the multiresolution representation, the cost of step 3) scales nearly linearly with system size, but has a large prefactor and can dominate for small systems, especially at high accuracy. Step 4) also scales close to linearly but has a small prefactor. Step 5) has an effective scaling between linear and quadratic, but with a relatively small prefactor as long as all but $O(1)$ of the orbitals are strongly localized. For large systems, step 2) dominates since construction of both the potential and kinetic matrices are observed to have a close to quadratic scaling with significant prefactor. In this work we have chosen to focus on computation of the kinetic energy matrix for two reasons — it tends to be most expensive and it is also the most complex in both data motion and implementation. Progress on this term for both performance and programmer productivity represents a significant advance in the state of practice.

### 7.5.2 Platform Description

$m_1$ is a performance bottleneck operation for very large scale KEMC calculations. To run at that scale, it requires a significant amount of memory. To test this benchmark, we ran it on a single large-memory node configured with 96 x 32 Gigabyte DDR4 Memory Modules operating at 1,600 Mega-Transfers/s, totaling 3 Terabytes of RAM (3,072 Gigabytes). This system has 4 Intel E7-8870v3 processors with 18 cores, each operating at 2.1 Gigahertz, for a total of 72 cores and 144 threads (via hyper-threading).
We ran \textbf{m2} and \textbf{M1} on a 100-node Infiniband cluster. Nodes are connected with high speed Infiniband with 30 nodes per IB switch. Each node has two Intel Xeon E5-2690v3 CPUs, with 24 cores at 2.6GHz, and a total of 128GB memory.

\textbf{7.5.3 Experimental Results}

\textbf{m1} \quad We generated two vectors of random Gaussian functions with 4096 functions in each vector for this experiment. We present strong scaling results for three different implementations of \textbf{m1}. The X-MADNESS-Fused version treats each inner product as an individual operator, and fuses $N \times N$ operators. The MADNESS implementation identifies the permutability of the traversal loop as described in Sec. 7.4.3, but it treats the transpose conjugate as a black box function call. The X-MADNESS-DGEMM, version performs the loop permutation, along with affine code transformation to represent the computation at each node label as a DGEMM. The results are shown in Fig. 7.4.

\textbf{m2 and M1} \quad We generated vectors of randomly generated Gaussian functions with 16 and 8 functions, respectively in each vector for this experiment. We present strong scalability for \textbf{m2} and \textbf{M1} and weak scalability results for \textbf{m2} using a base MADNESS implementation and one using the X-MADNESS DSL. The MADNESS implementation simply executes each operator in sequence, one after another, with global barriers between dependent operators. The X-MADNESS DSL implementation fuses all operators together. The results for \textbf{m2} are shown in Fig. 7.6, and results for \textbf{M1} are shown in Fig. 7.5.
Figure 7.4: Strong Scalability for MatrixInner (m1)

Figure 7.5: Strong Scalability for Kinetic Energy Matrix Calculations (M1)
We do not show weak scalability results for m1 and M1 benchmarks because we were not able to run the problem sizes required to test weak scalability, due to insufficient memory on the cluster.

7.5.4 Discussion

m1: While all three versions are optimized by recognizing that the traversal loop can be permuted, only the X-MADNESS-DGEMM version transforms the computation at each node label into a dense matrix-multiply as described in Sec. 7.4.3. By doing so we enable much better reuse of data at the various levels of the memory hierarchy, benefiting from tuned vendor implementation of the BLAS functions.

M1 and m2: For the weak scalability experiment (m2), we increased the number of functions proportionally with increase in number of nodes. As the functions are randomly generated, the k-d trees used for representing these functions are structurally different for each function, and the amount of computation for computing the derivative operator is not exactly the same for all functions. Thus, there is some variance in the amount of work per core introduced by the structural differences in the k-d trees as we increase the number of cores, despite a proportional increase in the number of functions. This accounts for the up and down trends in the weak scalability results. However, this does not affect the relative performance between MADNESS and X-MADNESS, since we use the same input for both.

The performance improvement with the X-MADNESS DSL over the MADNESS DSL in the m2 and M1 benchmarks come primarily from the following four factors: i) improved data locality, ii) reduced number of global fences, iii) improved coarse
Figure 7.6: Strong and Weak Scalability for m2 Derivative Sequence (m2)

- Grain parallelism, and iv) reduced task generation due to a single tree traversal.
We compare the performance characteristics of the MADNESS and X-MADNESS DSL (denoted “fused”) for each of these benchmarks in Table 7.1. Coarse grain parallelism denotes the number of operators executed for each task spawned by the MADNESS runtime. Global barriers shows the number of global fences required by the benchmark. Fusion enabled reuse shows the potential for improved locality through fusion. If an operator is a producer for \( m \) different consumer operators, then the fusion enabled reuse for the intermediate produced by the producer is \( m \). We show the average reuse in Table 7.1. Finally, we show the total number of tree traversals executed for each benchmark as a function of the size of the input vector of functions.

To compute \( m^2 \), MADESSS requires three global barriers, one at the end of each derivative operation, while the X-MADNESS (Fused) version requires only one at the end. Additionally, there are producer-consumer relations between computation of derivatives along \( x \) and \( y \), and \( y \) and \( z \). The fused schedule improves data locality with these intermediates. Finally, the fused version requires a single tree traversal to compute all the derivatives, while the MADNESS version requires a tree traversal for each derivative for each function in the vector (1 vs \( 3*N \), where \( N \) is the size of the vector).

The performance improvement in \( M1 \) can be explained in similar terms, as shown in Table 7.1. Through these improvements, the performance with the X-MADNESS DSL is even faster than the execution time of individual components of \( M1 \), such as the derivative operation using the MADNESS implementation.
Table 7.1: Performance Characteristics of m2 and M1

<table>
<thead>
<tr>
<th>Performance Characteristics</th>
<th>MADNESS m2</th>
<th>Fused m2</th>
<th>MADNESS M1</th>
<th>Fused M1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse grain parallelism</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2.33</td>
</tr>
<tr>
<td>Global barriers</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Fusion enabled reuse</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3.75</td>
</tr>
<tr>
<td>Number of tree traversals</td>
<td>3N</td>
<td>1</td>
<td>14N</td>
<td>1</td>
</tr>
</tbody>
</table>

7.6 Related Work

DSLs (Domain Specific Languages) offer a promising alternative to general-purpose languages for enabling high productivity, high performance, as well as portability across different parallel systems. Two examples of DSLs that have enabled high productivity and high performance for significant production application codes are STELLA [54] for the COSMO weather modeling code and the TCE (Tensor Contraction Engine) [59, 122] for the NWChem [131, 1] computational chemistry code. This chapter describes a DSL (Domain Specific Language) and optimizing compiler that has been developed to enable performance improvement and to ease the porting of the production MADNESS framework to future parallel systems.

Many of the concepts in MADNESS were motivated by the success of other frameworks such as Cilk [14] and Charm++ [73]. But unlike Cilk, MADNESS works portably in a distributed memory environment. Unlike Charm++, MADNESS is fully compatible with legacy software, which is a nearly fundamental constraint in scientific computing. Other projects that target distributed-memory parallel systems include ACES [125, 10, 124, 113], a parallel quantum chemistry suite that implements dense tensor computations, and STAPL [8], an extensible parallel graph library for
multi-core CPUs and distributed systems. STAPL does not currently use any compiler optimizations. Another project is the Galois system [79, 102]. It includes a programming model and a runtime component that uses speculative parallelization to extract parallelism from irregular computation, but does not focus on any data structure centric optimization for computations on spatial trees.

Burstall and Darlington presented a transformation system for recursive programs [17]. Nandivada et al. [94] presented transformations to optimize task parallel programs. Neither approach considers fusion or takes data locality into account.

Regarding tree traversal optimization, Jo and Kulkarni [71] exploit access information to optimize data locality across concurrent pre-order tree traversals. Meyerovich et al. [93] consider the synthesis of parallel fused tree-traversal schedules for attribute grammars, which exhibit simpler access patterns. Weijiang et al. [137] use a hybrid compile-time/runtime approach to analyze recursive pre-order tree traversals to perform locality- and parallelism-enhancing transformations. These approaches are superseded by the more general FuseT framework by Rajbhandari et al. [106], which we build upon and significantly extend in several ways in the present work: addressing a broader class of dependencies, incorporation of several domain-specific optimizations, and deployment in the production distributed-memory MADNESS framework.

7.7 Conclusion

The MADNESS (Multiresolution Adaptive Numerical Environment for Scientific Simulation) system [90] provides a general purpose numerical environment for a very broad set of applications, roughly characterized by the solution of differential and integral equations in multiple dimensions using adaptive meshes and fast solvers on
trees. However, the current implementation of MADNESS faces scalability challenges due to high overheads for dynamic synchronization and data movement.

In this chapter, we have presented a layered domain-specific compiler for MADNESS, framed as an embedded DSL system. We introduced a MADNESS-specific C++ embedded DSL system called X-MADNESS to describe operator specifications, including neighbor dependences and sequences of operators that can be analyzed and optimized by the X-Compiler. X-Compiler, the automatic compiler for X-MADNESS, implements key optimizations such as fusion of operator traversals, elimination of temporaries, and loop optimizations. Additionally, X-MADNESS provides runtime support for these optimizations. Using X-MADNESS, we have demonstrated up to 4x performance improvement over MADNESS on 2160 cores.
Chapter 8: Future Work

In this chapter, we will discuss potential research directions for further enhancing the performance and scalability of the applications and computation domains discussed earlier in this thesis.

8.1 Parallelizing Tensor Contractions

Quantum Chemistry methods such as the Coupled Cluster[11, 32] can require computing hundreds of tensor contractions in each iteration of the computation. The solutions we have developed in Chapters 2 and 3 allows us to compute each tensor contractions efficiently on a given distributed memory torus. However, computing each contraction one after another on the entire torus may not be the most optimal approach. This is because of two reasons: i) opportunities for parallelization, ii) varying performance and scalability of contractions.

The DAG of tensor contractions that must be computed in each iteration can be partitioned into phases where contractions in each phase can be executed in parallel. In addition, contractions within a phase can vary in dimensionality of the tensor, size of the iteration space, as well as in the number of external and internal iterators/indices. Due to this not all tensor contraction have the same cost, and they do
not achieve the same performance and scalability. For example, a tensor contraction with cost $n^6$ will scale much better than a tensor contraction with cost $n^3$ on a very large distributed memory torus. This is simply because the latter does not have enough work.

It maybe possible to enhance scalability and performance of the coupled cluster method by executing tensor contractions within a phase in parallel. Research in this direction would entail figuring out how to partition a distributed memory torus, and how to map different contraction within each phase of the coupled cluster iteration on to this partition so that the total execution time is minimized.

In this scheme, tensor contractions with large cost could be mapped to large partitions while smaller contractions would be mapped to smaller partitions. As scalability generally suffers as the size of the distributed cluster becomes too large compared to the problem size, this tensor specific partitioning would allow for high efficiency without running into scalability issues.

8.2 Convolution Neural Network

A potential extension to the work on CNN presented in this dissertation would be to study performance tradeoffs using different convolutions approaches from a data movement lower bound perspective. The work presented in this thesis shows performance tradeoffs using unfold+GEMM vs other more direct approaches optimized using the framework developed in this thesis. A third approach for computing convolutions is using fast fourier transform (FFT). Convolutions in spatial domains are transformed into pointwise multiplications in the frequency domain. However, there is a cost associated with computing the FFT. While the computational cost of FFT
based approach and non-FFT approaches can be compared trivially, a more in depth analysis is required to understand the associated data movement lowerbounds associated with FFT based approaches. It is possible that for some convolution kernels, an FFT based approach may incur more data movement that a non-FFT based approach even though FFT based approach maybe more favourable computationally.

### 8.3 Recursive K-d tree traversal in MADNESS

Both in XMADNESS and FuseT, the computation that must be performed for each recursive tree traversal operator at a node label is defined using a black box ‘compute’ method. This means that any optimization across operators cannot be performed even with fusion. In other words, a fused schedule simply executes the ‘compute’ methods for each operator in the fused schedule at every relevant node label. A potential future research direction is to explore if it is possible to expose the individual computations of each operator, thereby allowing inter-operator optimizations in fused schedules.
Chapter 9: Conclusion

On modern computing systems, data bandwidth can be several orders of magnitude slower than the computation throughput. This limited data bandwidth is one of the major causes of limited performance and scalability for a wide range of applications on modern computing systems.

To address this challenge, we developed locality optimization based solutions for various regular and irregular applications, and computation domains. We showed that our solutions can enhance performance and scalability of Tensor Contractions, Four-Index Transform, Convolutions Neural Networks, and Recursive Tree Traversal programs.

Locality enhancing solutions for computational domains such as Tensor Contractions and Recursive Tree Traversals can impact a large number of scientific applications that use these type of computation. This is a small step in solving the limited data bandwidth challenge. However, by continuing to identify computation domains that are widely used across multiple applications, it may be possible to develop a handful of locality based solutions with wide applicability, bringing us closer to overcoming the limited data bandwidth challenge for all applications of interest.
Bibliography


[41] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data access complexity of programs. In


[95] Jarek Nieplocha and Robert Harrison. Shared memory programming in meta-
computing environments: The global array approach. *The Journal of Super-

[96] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan,
Harold Trease, and Edoardo Aprà. Advances, applications and performance
of the global arrays shared memory programming toolkit. *Int. J. High Per-

[97] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan,
Harold Trease, and Edoardo Aprà. Advances, applications and performance
of the global arrays shared memory programming toolkit. *Int. J. High Per-

nwchem tce for different communication patterns. *4th International Workshop
on Performance Modeling, Benchmarking and Simulation of High Performance
Computer Systems (PMBS13)*.

[99] J.C. Pei, G.I. Fann, W. Nazarewicz, R.J. Harrison, J. Hill, D. Galindo, and
J. Jia. Coordinate-space hartree-fock-bogoliubov for superfluid fermi systems

[100] J.C. Pei, M. Stoitsov, G. Fann, W. Nazarewicz, N. Schunck, and F. Xu. De-
formed coordinate-space Hartree-Fock-Bogoliubov approach to weakly bound

[101] Liu Peng, Richard Seymour, Ken-ichi Nomura, Rajiv K Kalia, Aiichiro Nakano,
Priya Vashishta, Alexander Loddoch, Michael Netzband, William R Volz, and
Chap C Wong. High-order stencil computations on multicore clusters. In *Par-
allel & Distributed Processing, 2009. IPDPS 2009. IEEE International Sympo-

[102] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Am-
ber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman

[103] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ra-
manujam, P Sadayappan, and Nicolas Vasilache. Loop transformations: con-
vexity, pruning and optimization. In *ACM SIGPLAN Notices*, volume 46, pages


